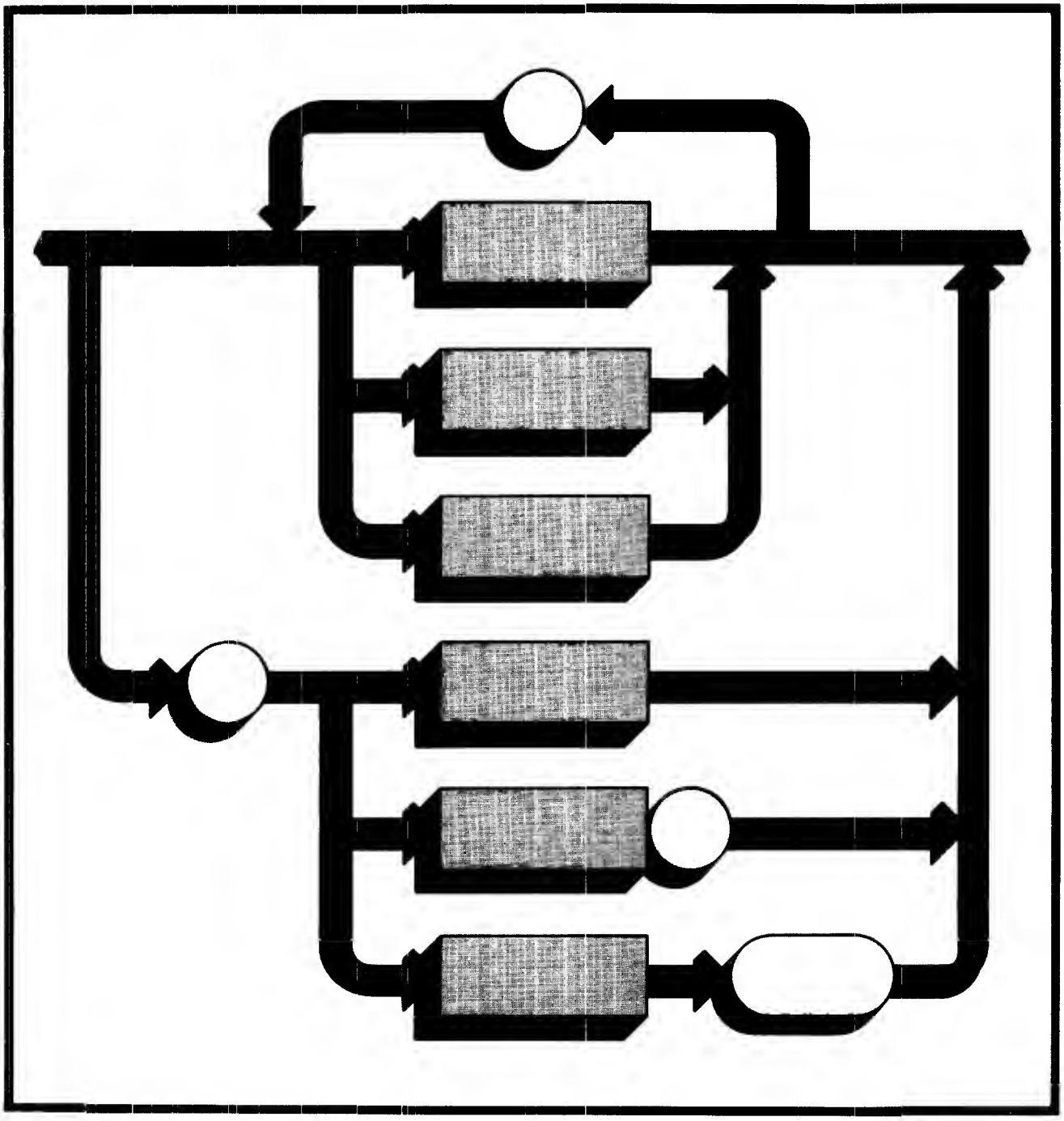
HP 9000 Series 200/300 Computers

HEWLETT PACKARD

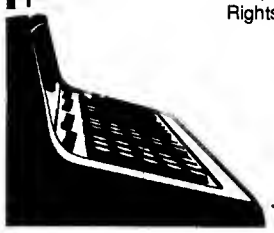
Pascal 3.1 Workstation System

Vol. II: Programming and Configuration Topics



Pascal 3.1 Workstation System
Vol. II: Programming and Configuration Topics
for the HP 9000 Series 200/300 Computers

Manual Reorder No. 98615-90022



Hewlett-Packard Company
3404 East Harmony Road, Fort Collins, Colorado 80525

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

May 1985...Edition 1

Table of Contents

Overview of Workstation Software Features

Chapter 10

Introduction

This chapter briefly lists the features of the Pascal language implemented on the Series 200 Workstation System. It also briefly describes the Procedure Library supplied with the system.

Chapter Contents

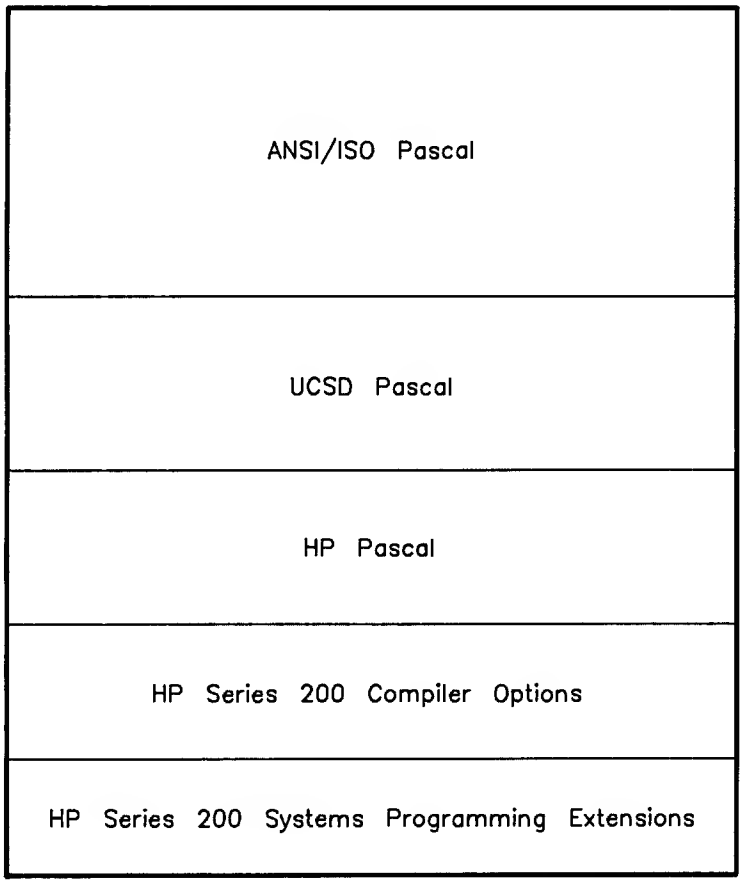
- Constituents of the HP Series 200 Pascal language implementation
- ANSI/ISO Pascal features
- UCSD Pascal[1] extensions
- HP Pascal extensions
- HP Series 200 Pascal Compiler options
- HP Series 200 Systems Programming extensions
- Overview of HP Series 200 Workstation software libraries

1 UCSD Pascal is a trademark of the Regents of the University of California.

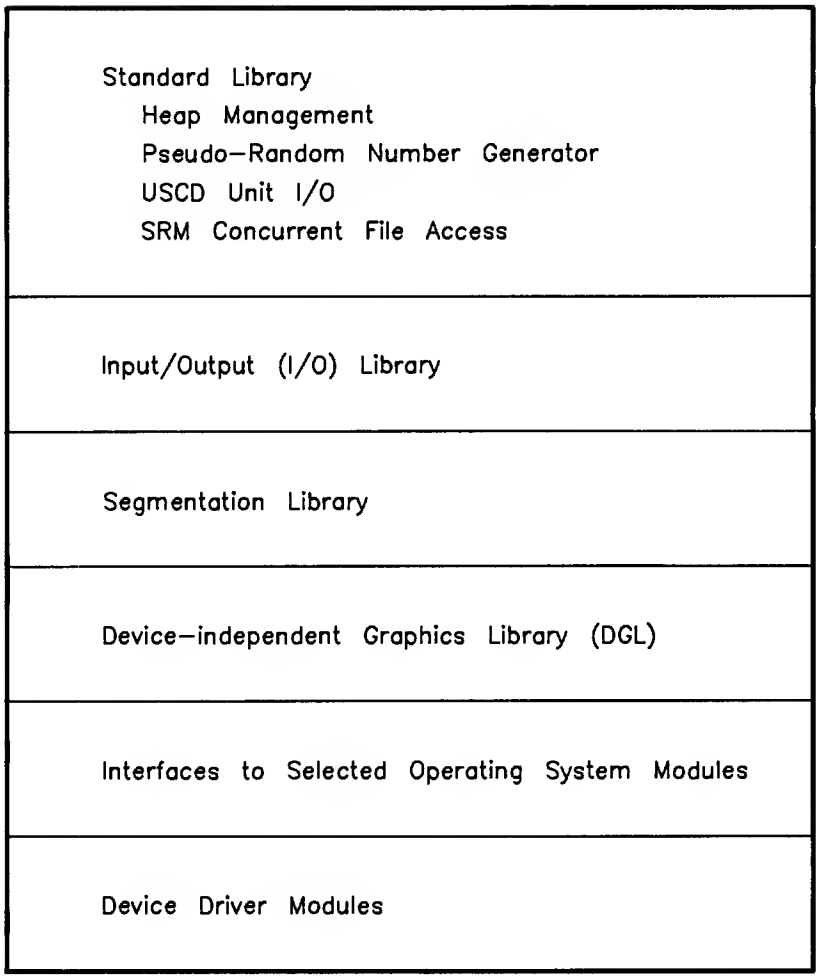
The Big Picture

The software supplied with the HP Series 200 Workstation Pascal system can be divided into several constituent parts, as shown in the following diagrams:

Constituents of the Series 200 Implementation of Pascal



The HP Series 200 Software Libraries

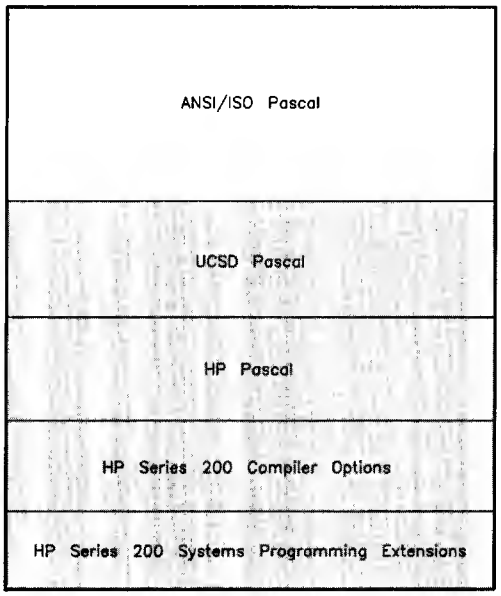


Subsequent sections of this chapter further describe each part of the drawings.

The Series 200 Implementation of Pascal

The HP Series 200 Workstation implementation of the Pascal language contains a full complement of features. This section describes the constituents of this implementation of the Pascal language.

ANSI/ISO Pascal



The term "ANSI/ISO" is an abbreviation of for two different Pascal standards. The "ANSI" portion stands for the Pascal standard adopted jointly by ANSI (the American National Standards Institute) and IEEE (the Institute of Electronics and Electrical Engineers). The "ISO" portion stands for the Pascal "Level 1" standard adopted by ISO (the International Standards Organization).

The HP Series 200 Workstation Pascal implementation contains **all** of the features of both the ANSI/IEEE and the ISO Pascal standards. Programming in ANSI/ISO Pascal is described in the *Programming and Problem Solving with Pascal* textbook supplied with the Workstation Pascal system.

Here is a list of the keywords in ANSI/ISO Pascal, which are all supported in this implementation.

Declarative Statements

```
program
const
label
type
var
procedure
function
```

Program Parameters

```
input
output
```

Program-Flow Control

```
begin...end
case...of
if...then...else
goto
for...to
   ...downto
repeat...until
while...do
```

Standard Procedures

```
get
new
pack
page
put
read
readln
reset
rewrite
unpack
write
writeln
```

Data Types

```
array
boolean
char
file
integer
packed
real
record
set
text
with
```

Numeric Functions

```
abs
arctan
cos
exp
ln
odd
round
sin
sqr
sqrt
trunc
```

Set Operators

```
+
-
*
in
```

Pre-defined Constants

```
false
true
nil
maxint
```

Ordinal Functions

```
chr
ord
pred
succ
```

File Functions

```
eof
eoln
```

Assignment Operator

```
:=
```

Arithmetic Operators

```
+
-
*
/
div
mod
```

Comparison Operators

```
<
<=
=
>=
>
<>
```

Logical Operators

```
and
not
or
```

Extending "Standard" Pascal's Capabilities

Pascal is a general-purpose programming language. It was originally designed as a language to teach structured programming, and it has since gained widespread use due to this orientation. However, as with all languages, the Pascal programming language cannot satisfy *every* programmer's needs; in such cases, the feature set can be "extended" to fit certain applications.

There are two general ways to extend the capabilities of a programming language:

- Add extensions to the language itself.
- Write "library" routines that can be called from the language.

The Pascal Workstation designers have used *both* methods to add capabilities to this system. Language extensions are described in the following sections, followed by libraries in later sections.

Language Extensions

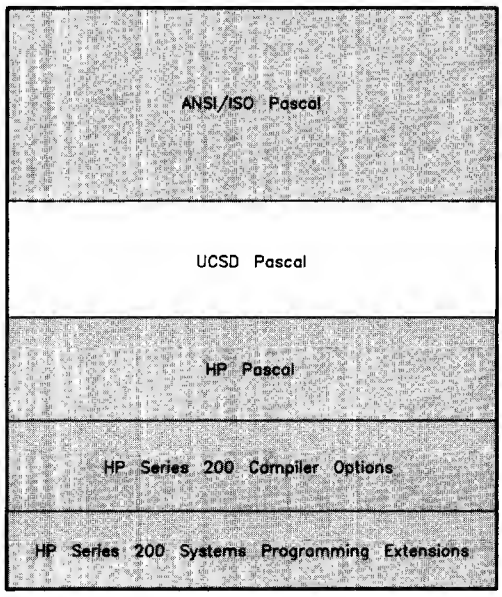
Adding extensions to a language requires that the designers add to the list of "keywords" that the Compiler will recognize. This Pascal implementation contains four general categories of extensions:

- UCSD Pascal[1] extensions
- HP Pascal extensions
- Series 200 Workstation Pascal Compiler options
- HP Series 200 Systems Programming extensions

Each category is further described in subsequent sections.

1 UCSD Pascal is a trademark of the Regents of the University of California.

UCSD Pascal Features



UCSD Pascal adds many useful features to the "standard" Pascal language. The UCSD Pascal features which this Pascal implementation supports are fully described in "Supported Features of UCSD Pascal" in the "Workstation Implementation" appendix of the *HP Pascal Language Reference*. Here is a brief summary, showing the various levels of support for UCSD features.

Fully Supported

`blockread`
`blockwrite`
`close`
Compiler options
`external`
Files
`fillchar`
Heap Management
`moveleft`
`moveright`
Reals
`scan`
`set`
`sizeof`
Special Program Heading
Standard Units
Strings
`unitbusy`
`unitclear`
`unitread`
`unitwrite`
Untyped Files

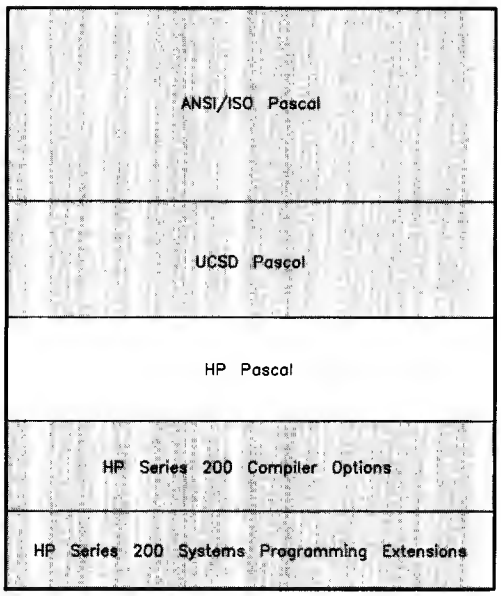
Unsupported Features

`log`
Long integers
Multi-word comparisons
`pwroften`

Slight Differences

`CASE`
Comments
Compilation Units
`exit`
`gotoxy`
`halt`
16-bit Integers
`interactive`
`ioresult`
`memavail`
`seek`
`time`
Type Checking
`unit`

HP Pascal Features

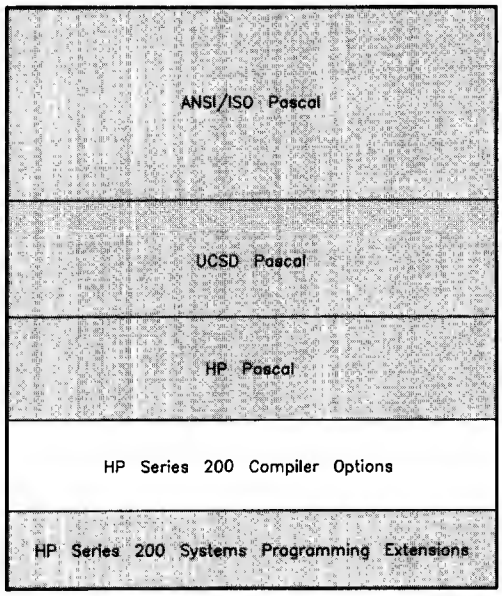


HP Pascal includes many of the UCSD extensions to ANSI/ISO Pascal, plus some of its own. The HP Pascal extensions to ANSI/ISO Pascal are briefly summarized in this section, and more fully described at the **beginning** of the *HP Pascal Language Reference*. Complete, detailed descriptions of individual procedures, reserved words, etc., are provided in the **body** of the same reference. Examples of using many of these HP Pascal extensions are provided in the subsequent "Programming Topics" chapters of this manual. Here is a brief summary of the areas in which HP Pascal has extensions:

HP Pascal Features

- Compiler Options
- Conformant Arrays
- Constant Expressions
- Early Program Termination
- Extended Variable Assignment Compatibility
- Full File-I/O Feature Set
- Functions May Return Any Structured Type
- Heap Management Capabilities
- Identifiers May Contain "-"Character
- Intermixing of Declaration Parts of Programs
- Longreal Data Type
- `minint` Pre-defined Constant
- Modules
- Numeric-to-String Conversions
- `OTHERWISE` in CASE Statement
- Record List in `WITH` May Include Function Calls
- Record Variants May Be Subranges
- String Literals May Contain Control Characters
- String Data Type
- Structured Constants

Series 200 Workstation Pascal Compiler Options



Some Compiler options affect the way that the Compiler emits object code, while others allow the use of UCSD and HP Series 200 Systems Programming extensions. For a description of each option, refer to the "Series 200 Compiler Options" section of the "Workstation Implementation" appendix of the *HP Pascal Language Reference*.

Code-Generation Control

```
callabs
code
code_offsets
debug
float_hdw
heap_dispose
if
iocheck
ovflcheck
partial_eval
range
stackcheck
```

Compiler Listing Control

```
linenum
lines
list
page
pagewidth
tables
```

Message Control

```
ansi
copyright
warn
```

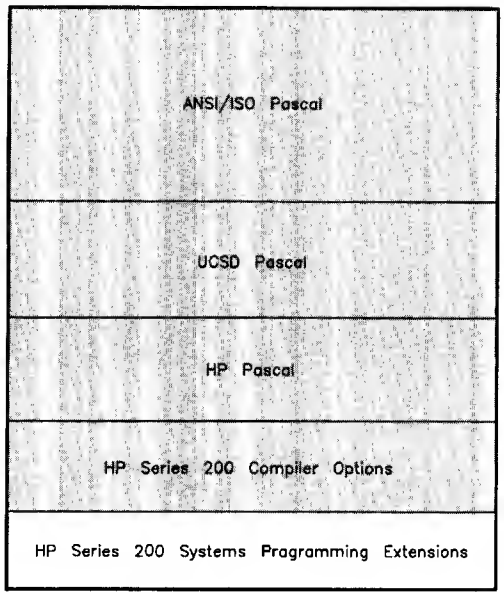
Use of External Files

```
def
include
ref
search
search_size
```

Language Feature Control

```
alias
allow_packed
save_const
switch_strpos
sysprog
ucsd
```

HP Systems Programming Extensions



The HP Series 200 Systems Programming extensions are briefly described in the following list. Programming examples of most features are given in the "Programming Topics" chapters of this manual. Complete descriptions of all features are provided in the "Systems Programming Extensions" section in the "Workstation Implementation" appendix of the *HP Pascal Language Reference*.

Error Trapping and Simulation

`escape`
`escapecode`
`ioresult`
`try/recover`

Absolute Address of Variables

`addr`
`var` syntax

Determining Size of Variables and Types

`sizeof`

Relaxed Type-Checking

`anyptr`
`anyvar`

Special Procedure Calls

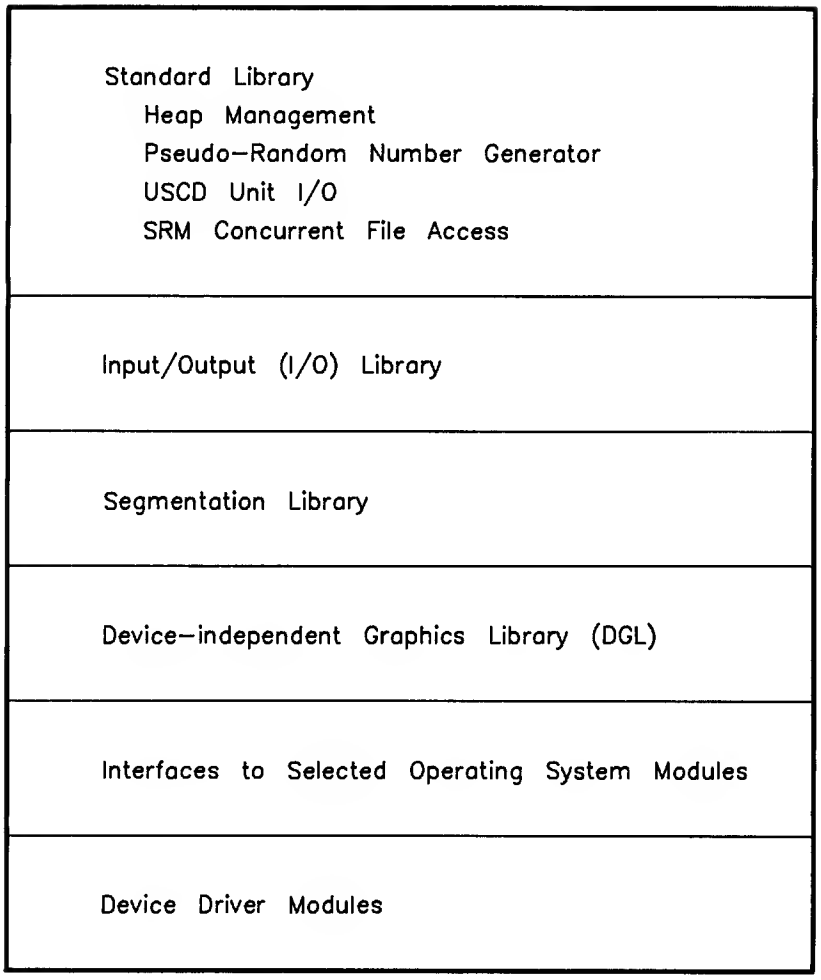
`call`
Variables of type `procedure`

With the power of these System Programming features, however, comes the restriction that programs that use them will probably be dependent upon the Workstation Operating System and possibly the hardware on which the programs are executed (which may include its specific configuration).

A Final Word Concerning Language Extensions

Although these extensions provide many additional capabilities to the Pascal language, they do not provide a full set of tools for accessing Workstation computer capabilities. That tool set is provided by software libraries.

HP Series 200 Software Libraries



The second way to "extend" a language's capabilities is to place commonly used procedures, functions, data types, and so forth into "libraries" which are accessible to all programmers on the system. In this system, these libraries consist of object-code "modules" produced by the Compiler, Assembler, or Librarian. Each module is an *independent* program fragment that contains data and procedures which are usable by other programs (and other modules). The general topic of modules is discussed in the Compiler, Assembler, and Librarian chapters of this manual.

Library Modules

The following list of libraries is organized according to the file in which they are shipped with the system. A list of the discs and files upon each is provided in the *Pascal User's Guide*; you may also want to generate your own list by using the Filer's List (or Extended_list) command.

The LIBRARY file provides the following four modules:

- The HPM (Heap Management) module provides the new and dispose procedures that can be used to allocate and reclaim memory used by dynamic variables. See the "Dynamic Variables and Heap Management" chapter for examples.
- The RND (Random Numbers) provides the random procedure and the rand function that are used for generating pseudo-random numbers. See the "Numeric Computation" chapter for examples.
- The UIO (UCSD Unit I/O) module provides the blockread, blockwrite, unitbusy, unitclear, unitread, unitwait, unitwrite procedures that are used for "low-level" input and output (I/O) operations with mass storage "blocks". See "Supported Features of UCSD Pascal" in the "Workstation Implementation" appendix of the *HP Pascal Language Reference*.
- The LOCK module provides features that support concurrent file access on the Shared Resource Manager (SRM) "file server" system. The lock function and the unlock and waitforlock procedures are used to lock and unlock shared files. See the "Programming with Files" chapter for examples.

The IO (Input/Output) file provides several modules which provide constants, types, variables, procedures, and functions used for communicating through HP Series 200 interfaces. These procedures are described in several chapters of the *Pascal Procedure Library* manual. A functionally grouped list of all procedure and functions is also provided at the beginning of the "Procedure Reference" section of that manual.

The "DGL" (Device-independent Graphics Library) files contain the modules that provide procedures and functions for drawing and labeling graphics images on both raster and physical-pen plotting devices. They also contain procedures and functions for graphics input devices, such as a graphics tablet, mouse, knob, and TouchScreen™. See the *Pascal Graphics Techniques* manual for examples.

The INTERFACE file provides an interface to selected Operating System modules. Here are the modules which are documented in the manuals:

- IODECLARATIONS and various other modules provide many useful data structures that are used by various parts of the system and by many procedure libraries. See the "Introduction to I/O" chapter of the *Pascal Procedure Library* manual for details on the IODECLARATIONS module.
- The SYSDEVS (System Devices) module provides procedures and functions for using the built-in displays, keyboards, and timers of Series 200 machines. These procedures are described in the "System Devices" chapter of the *Pascal Procedure Library* manual.
- The SEGMENTER module provides procedures and functions for executing small segments of larger programs, in order to decrease memory requirements. These procedures are described in the "Segmentation" chapter of the *Pascal Procedure Library* manual.

Note

Of these Operating System modules with interfaces in the INTERFACE module, *only* the use of the IODECLARATIONS, SYSDEVS, and SEGMENTER modules are documented in the *Pascal Procedure Library* manual. Descriptions of the other Pascal 3.0 Operating System modules are given in the *Pascal System Internals Document* (version 3.0 or later).

Other files on the CONFIG: disc provide device driver modules which contain code that the system uses to communicate with interfaces and devices. For instance, the GPIO file (which contains a module of the same name) provides driver routines that are used to communicate through an HP 98622 General-Purpose Input/Output (GPIO) interface. Note that these device driver files contain no "export text" that describes the procedures, etc. in the modules, because the drivers don't need it.

The most commonly used modules are automatically loaded into your system during the booting process, because they are in the INITLIB file on the BOOT: disc. For instance, the CS80 module provides routines which communicate with CS/80 and SS/80 type disc drives. The "Special Configurations" chapter of this manual describes the booting process and all driver modules shipped with the system.

You may have had to install other driver modules yourself while configuring your system. The description of this process is in the "Adding Peripherals" section of the *Pascal User's Guide.*

User-Designed Modules

One of the most powerful capabilities of this system is that you can design your own specialized libraries using the HP Pascal `module` construct. That subject is discussed in the "Compiler," "Assembler," and "Librarian" chapters of this manual, as well as in the "Overview" chapter of the *Pascal Procedure Library* manual.

Notes

Chapter 11 Data Structures

One of the most powerful features in Pascal is the ability to create *data structures.* A data structure is an arrangement of types of data in such a way that it most accurately represents the model you are trying to represent.

Data Types

Hewlett-Packard Workstation Pascal supports all standard Pascal data types. This section briefly summarizes these types and how they are used. Extensions to standard Pascal provided by Workstation Pascal will be noted in the text.

Scalar Types

The word "scalar" in the phrase "scalar types" means *single-valued;* that is, variables of these types each contain only one piece of data. This is opposed to the concept of "structured" types, a kind of do-it-yourself data type. Structured types are covered later in the chapter.

Standard Data Types

The simplest data structures are those simple, standard types provided by the Pascal language. These are:

`integer`	A 32-bit signed integer number.
`real`	A 64-bit signed floating-point number.
`char`	An 8-bit ASCII character.
`boolean`	True or false values.

Note that the first three types above are implementation-dependent in the areas of number of bits, format of bits, etc.

HP Pascal Features

Array Constants

To make a constant which is of an array type, you specify the type, a "[", the values, separated by commas, and a "]". The base type must be declared before the constant declaration; e.g., in the sample code below, you must declare a type `MonthsType` before you can declare the 12-element constant array `DaysPerMonth`. For example:

```
type
  MonthsType=          array [1..12] of integer;
const
  DaysPerMonth=        MonthsType[31,28,31,30,31,30,31,31,30,31,30,31];

type
  LineType=            array [1..4] of real;
  MatrixType=          array [1..4] of LineType;
const
  Identity=            MatrixType[LineType[1.0, 0.0, 0.0, 0.0],
                                  [LineType[0.0, 1.0, 0.0, 0.0],
                                  [LineType[0.0, 0.0, 1.0, 0.0],
                                  [LineType[0.0, 0.0, 0.0, 1.0]];
```

Note

When making a structured constant of a multidimensional array, it must be declared one dimension at a time; e.g., a vector of vectors, rather than a 2D array.

When declaring an array constant and there are several identical values in consecutive places, you can declare them something like this:

```
type
  VectorType=          array [1..100] of integer;
const
  Vector=              VectorType[1,2,3,6 of 0,7,90 of 0];
```

This results in an array, none of whose elements' values can be changed, in which there are these values:

1, 2, 3, six zeroes, 7, and ninety more zeroes.

Record Constants

When declaring a constant of some record type, you must specify the field name before the corresponding value. For example:

```
type
  DateType=           record
                        Month:     1..12;
                        Day:       1..31;
                        Year:      integer;
                      end;
  MaritalStatusType= (Single, Married, Separated,Divorced, Widowed);
  IntervieweeType=    record
                        Name:              string[30];
                        BirthDate:         DateType;
                        MaritalStatus:     MaritalStatusType;
                        NumberOfChildren: 1..20;
                        Education:         set of (HighSchool, BA, BS, MA,
                                                   MS, PhD, DD);
                      end;
const
  ThisPerson=         IntervieweeType[Name:             'John Q. Public',
                                      BirthDate:        DateType[Month: 10,
                                                                 Day:   20,
                                                                 Year:  1955],
                                      MaritalStatus:    Married,
                                      NumberOfChildren: 1,
                                      Education:        [HighSchool, BS]];
```

Set Constants

Set constants can be made. In a set constant, the elements must be surrounded by brackets, so the compiler knows that it is a set constant. No base type is required in order to declare a set constant:

```
const
  Vowels=             ['a','e','i','o','u'];  {set constant}
```

Packing Variables

In the discussion of arrays, there was some mention of PACs, or packed arrays of characters. Arrays of ⟨*type*⟩ are different than packed arrays of ⟨*type*⟩, no matter what ⟨*type*⟩ is.

Probably the most common reason for packing a data structure is that it takes less memory to contain the data. However, you pay for the reduced memory demands in *increased* time required to access the variables. There is a way, though, to get the best of both worlds: the lower memory consumption of packed variables and the high speed of unpacked variables. The `pack` and `unpack` procedures allow you to do this.

Suppose you have an array of packed variables. They are packed to save file space and memory space. You can unpack an element, process it at the higher speed afforded by unpacked variables, and repack it.

One precaution: when packing variables, you may not get exactly what you wanted. The compiler may do some field justification to byte- or nybble boundaries in order to make processing faster.

```
$tables$
   . . .
type
  NotReallyPackedRec=   packed record
                          case integer of
                            1: (RealNumber:  real);
                            2: (SignBit:     boolean;
                                Exponent:    packed array [1..11] of boolean;
                                Mantissa:    packed array [1..52] of boolean;)
                        end;
```

One would think that this is a convenient way to deal with the various subfields in a real number. However, although you specify "`packed`", it is not really packed; it's no more compact than if you hadn't specified `packed`. You can verify this by specifying the compiler option `$tables$`, which causes the following information (among other things) to be printed in the listing:

```
NOTREALLYPACKEDREC type
 record unpacksize=11 align=2
   EXPONENT          field offset=2
   MANTISSA          field offset=4
   REALNUMBER        field offset=0
   SIGNBIT           field offset=0 bitoffset=0
```

The reason that the above example is not any more compact than the unpacked version is that there are some constraints on packing. While an attempt is made by the compiler to use less space, there are also some requirements for efficient access; both packing *and* alignment are considerations:

- Nothing whose size is a long word or more is packed. Thus, integers, pointers, and reals are not packed. Also, if a record contains other records, the internal records are not packed with respect to the record which contains them.
- Everything that is packed must be accessible with one long-word access, and long-word accesses must take place on even-byte boundaries. For example, it is conceivable that a 17-bit field would not be accessible by a single 32-bit access. Thus, this field would not be packed in this way.
- Arrays are packed along 1, 2, 4, 8, or 16-bit boundaries. Thus, an array of 5-bit fields would be packed only to 8-bit boundaries.

In addition to the optional keyword `packed` in front of `array`, `record`, `set`, and `file` type specifiers, there are two routines, `pack` and `unpack`, which convert an `array of` ⟨*type*⟩ to a `packed array of` ⟨*type*⟩ and vice versa. See the *HP Pascal Language Reference* for details on `pack` and `unpack`.

Determining the Size of Variables and Types

The size (in bytes) of a data type or variable can be determined by the Systems Programming function[1] `sizeof`. To use this feature, you must use the `$sysprog$` or the `$ucsd$` Compiler option. Here are examples of usage:

```
$sysprog$
  . . .
VBytes:=sizeof(Variable);
TBytes:=sizeof(TypeName);
```

If the variable or type is a record with variants, optional tagfield constant(s) may follow the variable name parameter. Syntactically, it is similar to a call to the standard Pascal procedure `new`:

```
NBytes:=sizeof(RecVar,TrueField,BlueField);
```

1 Although `sizeof` looks like a function, it really is not one; it is actually a form of compile-time constant.

Absolute Addressing of Variables

Systems Programming extensions also provide the capability of programmatically setting and determining the absolute address of variables. This capability requires `$sysprog$`.

Setting a Variable's Absolute Address

A variable may be declared as located at an absolute or symbolically named address:

```
var
  SysFlag[hex('FFFED2')]:                    char;
  AssemblerSymbol['external_name']:          integer;
```

Each variable named in a declaration may be followed by a bracketed address specifier. An integer constant gives the absolute address of the variable. A quoted string literal gives the name of a load-time symbol which will be taken as the location of the variable; such a symbol must be present in RAM when the program is run. These symbols are not accessed as globals and do not count against the 32K-byte limit per module.

Determining a Variable's Absolute Address

The `addr` function returns the address of a variable in memory as a value of type `anyptr`. This also requires `$sysprog$`.

```
type
  SomeType=          ⟨type_declaration⟩;
var
  Pointer1,Pointer2: anyptr;
  Variable1:         integer;
  Variable2:         SomeType;
      .
      .
      .
  Pointer1:=addr(Variable1);
  Pointer2:=addr(Variable2,Offset);
```

The `addr` function accepts, as an optional second parameter, an integer "offset" expression which will be added to the address; this has the effect of pointing "offset" bytes away from where the variable begins in memory. A positive offset addresses bytes higher in memory, and a negative offset addresses bytes lower in memory.

The `addr` function is primarily used for building or scanning data structures whose shapes are defined at run-time rather than by normal Pascal declarations.

Note

Programs using this feature must be *very carefully debugged.* Careless use of the pointers returned by `addr` can crash your system.

The `addr` function has the same dangers described above for `anyptr`s, in addition to some of its own. Use of the "offset" can produce a pointer to almost anywhere, with concomitant dangers to the integrity of system memory.

Never use `addr` to create pointers to the local variables of a procedure or function. Storage for local variables is recovered when the routine exits, so the value returned by `addr` is ephemeral.

Conformant Arrays

Conformant arrays are arrays in a called routine which automatically conform to the size of the array which was passed to the routine. The conformant array feature allows arrays of various sizes to be passed to a single formal parameter of a routine. It also provides a mechanism for determining at runtime the indices with which the actual parameter was declared.

Conformant arrays are defined within the formal parameter list of a procedure or function. They may be passed by value or by reference.

Conformant arrays may be packed or unpacked. Their organizations, or representations are defined by "schemas." Unpacked schemas may have any number of indices, whereas packed schemas are limited to one index. In a schema with multiple indices, the final array definition may be either packed or unpacked. Conformant arrays may not be packed arrays of characters (PAC) types.

An abbreviated syntax is allowed for specifying multi-dimensional conformant arrays. The schema:

```
array [⟨index type⟩] of
  array [⟨index type⟩] of
    .  .  .
  array [⟨index type⟩] of ⟨type id⟩
```

can be written as:

```
array [⟨index type⟩;
       ⟨index type⟩;
          .  .  .
       ⟨index type⟩] of ⟨type id⟩
```

The bound identifiers (the low bound identifier and the high bound identifier in the index type specification) are used to determine the indices of the actual parameter passed to the formal conformant array. Their values are set when the routine is entered, and they remain constant throughout that activation of the routine.

Bound identifiers are special objects. They are not constants and they are not variables; thus, they cannot be used in `const` or `type` definitions, and may not be assigned to, or used in any other context in which a variable is assigned to (actual `var` parameter, `for`-loop control variable, etc).

Conformability

An actual array parameter must "conform" to the corresponding formal parameter. That is, an array variable may be passed to a routine with a corresponding formal conformant array parameter if the array variable's type "conforms with" the schema of the formal parameter.

An informal way of describing conformability is to say that the array variable's type conforms with schema if, for each dimension of array type and schema, the index types and component types of array type and schema "match."

For instance, given the following types and conformant array schemas:

Types:

```
type
  Index=      1..20;
  T1=         packed array [1..10] of integer;
  T2=         array [1..5, 1..10] of integer;
  T3=         array [1..50] of integer;
```

Conformant Array Schemas:

Schema 1: `array [lo..hi: Index] of array [smallest..largest: Index] of integer;`

Schema 2: `packed array [little..big: Index] of integer;`

Schema 3: `array [least..greatest: Index] of integer;`

Schema 4: `array [lo..hi: Index; lo2..hi2: Index] of integer;`

The following relationships are true:

- Type `T1` conforms with Schema 2 only.
- Type `T2` conforms with Schemas 1 and 4 only.
- Type `T3` does not conform with any of the schemas.

Equivalence
Two conformant array schemas are "equivalent" if all of the following are true:

- The ordinal type identifier in each corresponding index type specification denotes the same type.
- Either:
  - the type identifier of the two schemas denotes the same type, or
  - the component conformant array schemas of both schemas are equivalent.

Congruency
An actual array parameter of an actual procedure or function parameter must be "congruent" with the corresponding formal parameter. Two conformant array schemas are "congruent" if all of the following are true:

- The two schemas are both packed or unpacked.
- The two schemas are both by-value or by-reference schemas.
- The two schemas are equivalent.

An example of where you would be able to use conformant arrays to your advantage is shown below. Suppose you need a vector (a one-dimensional array) where the first element of the array equals 1, the second element equals 2, etc. With a procedure which uses conformant arrays, this might look like this:

```
var
   Vector1:      array [1..5] of integer;
   Vector2:      array [1..10] of integer;
   Vector3:      array [7..9] of integer;
      . . .
procedure DefineVector(var Vector: array [Lo..Hi: integer] of integer);
var
  I:             integer;
begin
for I:=Lo to Hi do
  Vector[I]:=I;
end;
      . . .
DefineVector(Vector1);
DefineVector(Vector2);
DefineVector(Vector3);
```

Any of the arrays, regardless of size, can be sent to the procedure `DefineVector`. In passing the array to the procedure, the bounds identifiers ("`Lo`" and "`Hi`") are defined. Inside the procedure, `Lo` and `Hi` can be used anywhere a variable or constant can be used, *except* in declaration statements. That is, you cannot declare another variable such as:

```
var
  NewArray:      array [Lo..Hi] of integer;   { Illegal! }
```

Nor can you "redimension"–change the size of–an array by assigning a value to a bounds identifier:

```
Lo:=3;        { Illegal! }
Hi:=4;        { Illegal! }
```

Nor can you do anything else to try to change such a value; such as pass it by reference to a procedure.

Another example of using conformant arrays is in multi-dimensional arrays. As usual in Pascal,

`array [`⟨*range*⟩`,` ⟨*range*⟩`] of` ⟨*type*⟩

is equivalent to

`array [`⟨*range*⟩`] of array [`⟨*range*⟩`] of` ⟨*type*⟩

Suppose you have defined a matrix thus:

```
type
  M4x4=          array [1..4, 1..4] of integer;
var
  M1:            M4x4;
```

You could define a procedure, using conformant arrays, to define the identity matrix:

```
procedure Identity(var Matrix: array [RowMin..RowMax: integer;
                                      ColMin..ColMax: integer] of integer);
var
  Row, Col:     integer;
begin
for Row:=RowMin to RowMax do
  for Col:=ColMin to ColMax do
    Matrix[Row, Col]:=ord(Row=Col);
end;
```

An additional legality check could be made to ensure that the matrix is square; a non-square identity matrix is a contradiction.

To send multiple conformant arrays to a procedure (or function; all these statements about conformant arrays can be applied to function parameters, too), you just separate them by semicolons in the usual way. Also, you can intermix conformant arrays passed by value and conformant arrays passed by reference[2]:

```
procedure MatMult(    Left:   array[LRowMin..LRowMax: integer;
                              LColMin..LColMax: integer] of integer;
                      Right:  array[RRowMin..RRowMax: integer;
                              RColMin..RColMax: integer] of integer;
                  var Answer: array[ARowMin..ARowMax: integer;
                              AColMin..AColMax: integer] of integer);
var
  Row, Col, Sum:        integer;
  I, J, K:              integer;
begin
if (LColMax-LColMin+1)<>(RRowMax-RRowMin+1) then
  begin
  writeln('For a matrix multiply, the number of rows in the left matrix');
  writeln('must equal the number of columns in the right matrix.  The');
  writeln('matrices passed to the matrix multiply routine failed this');
  writeln('test; resultant matrix zeroed.');
  for Row:=ARowMin to ARowMax do
    for Col:=AColMin to AColMax do
      Answer[Row,Col]:=0;
  end
else
  if (RRowMax-RRowMin+1)<>(RColMax-RColMin+1) then
    begin
    writeln('For a matrix multiply, the right matrix must be square.  The');
    writeln('right matrix passed to the matrix multiply routine failed this');
    writeln('test; resultant matrix zeroed.');
    for Row:=ARowMin to ARowMax do
      for Col:=AColMin to AColMax do
        Answer[Row,Col]:=0;
    end
```

2 If you pass a conformant array to a procedure, and, from that procedure, you wish to pass the array to another procedure, you must pass it (the second time) by reference.

```
  else
    begin
    for I:=LRowMin to LRowMax do
      for J:=LColMin to LColMax do
        begin
        Sum:=0;
        for K:=LColMin to LColMax do
          Sum:=Sum+Left[I,K]*Right[K,J];
        Answer[I,J]:=Sum;
        end;
    end;
end;
```

Notes

Chapter 12 Program Flow

Introduction

This chapter contains information on how you can alter the standard direction of program flow, which is normally one statement after another in sequential order. There are several different areas included in this chapter. They are:

- Standard Pascal branching,
- Procedure and function calls, and
- Procedure variables.

Standard Branching

All of the branching constructs available in standard Pascal are implemented in Workstation Pascal. The following list describes these constructs:

- `if/then/else`
- `for/do`
- `repeat/until`
- `while/do`
- `case/of`
- `goto`

These are described in the *Programming and Problem Solving With Pascal* book.

CASE/OF

HP Pascal supports these two extensions to the standard Pascal `case` statement.

- Subranges for `case` constant lists. For example, if you want the values 4, 5, 6, and 7 to cause the same action, you could type `4..7:` in the case constant list.
- The `otherwise` case. If none of the case constants match the value coming into the `case` statement, the `otherwise` clause, if it exists, will be executed. The `otherwise` clause consists of the word `otherwise` and one statement which may be compound. Note that there is no colon between the word "`otherwise`" and its statement.

See the *Pascal Language Reference* for more details on these extensions.

Procedures and Functions

HP Pascal incorporates all the standard parameter-passing rules which are in effect for standard Pascal. The following sections document only the HP extensions, and all require the compiler option `$SYSPROG$` to be in effect.

Relaxed Typechecking of VAR Parameters

The `anyvar` parameter specifier in a function or procedure heading relaxes type compatibility checking when the routine is compiled. This is sometimes useful to allow routines to act on a general class of objects. For instance, an I/O routine may be able to enter or output an array of arbitrary size.

```
$SYSPROG$                {required}
  . . .
type
  Buffer=                array [0..maxint] of char;
var
  Arr1:                  array [2..50] of char;
  Arr2:                  array [0..99] of char;
procedure Output_HPIB(anyvar Ary: Buffer; LoBound, HiBound: integer);
⟨procedure body⟩
    . . .
Output_HPIB(Arr1,2,50);
Output_HPIB(Arr2,0,99);
```

`Anyvar` parameters are passed by reference, not by value; that is, the address of the variable is passed. Within the procedure, the variable is treated as being of the type specified in the heading.

For instance, if an array of 10 elements is passed as an `anyvar` parameter which was declared to be an array of 100 elements, an error may very well occur. The called routine has *no way* of knowing what you actually passed, except perhaps by means of other parameters as in the example above. `Anyvar` should only be used when it's absolutely required, since it defeats the Compiler's normal type-safety rules.

Note

Programs calling routines with `anyvar` parameters should be *very thoroughly debugged!* Careless use of this feature can crash your system.

The ANYPTR Type

Another way to defeat type checking is with the non-standard type `anyptr`. This is a pointer type which is assignment-compatible with all other pointers, just like the constant `nil`. However, variables of type `anyptr` are not bound to a base type, so they *cannot* be de-referenced (i.e., `anyptr_var^` is not permitted). They may only be assigned or compared to other pointers. Passing as a value parameter is a form of assignment.

```
$sysprog$              {required}
  . . .
type
  Pointer1=            ^integer;
  Pointer2=            ^record
                         R1, R2:       real;
                       end;
var
  V1,V1a:              Pointer1;
  V2:                  Pointer2;
  AnyV:                anyptr;
  Which:               (Type1,Type2);
begin
new(V1);
new(V2);
. . .
if . . . then
  begin
  AnyV:=V1;
  Which:=Type1
  end
else
  begin
  AnyV:=V2;
  Which:=Type2
  end;
. . .
if Which=Type1 then
  begin
  V1a:=AnyV;
  V1a^:=V1a^+1;
  end;
end;
```

The compiler has no way to know if `anyptr` tricks were used to put a value into a normal pointer. If a pointer type which is bound to a small object has its value tricked into a pointer bound to a large object, subsequent assignment statements which dereference the tricked pointer may destroy the contents of adjacent memory locations.

Note

Routines that use the `anyptr` feature should be *very thoroughly debugged!* Careless use of this feature can crash your system.

Procedure Variables and the Standard Procedure CALL

Sometimes it is desirable to store in a variable the name of a procedure, and then later to call that procedure. For instance, the File System's "Unit Table" is an array which contains (among other things) the name of the driver to be called to perform I/O on each logical unit.

A variable of this sort is called a "procedure variable," or "procvar," for short. The "type" of a procedure variable is a description of the parameter list it requires. That is, a procedure variable is bound to a particular procedure heading.

```
$SYSPROG$    {REQUIRED for PROCVARS and CALL}
  . . .
TYPE
  ProcVar=           PROCEDURE(OP: INTEGER);
VAR
  P:                 ProcVar;
  I:                 INTEGER;
PROCEDURE Q(OP: INTEGER);    {identically structured parameter list}
  . . .
P:=Q;                {P gets the name of Q; in effect, P points to Q}
CALL(P,I);           {name of procvar, then appropriate parameter list}
```

A procedure variable is invoked by the standard procedure `CALL`, which takes the procedure variable as its first parameter, and a further list of parameters just as they would normally be passed to the procedure having the corresponding specification.

It is not possible to create a "function variable", that is, a variable which can hold the name of a function.

Don't assign the name of an inner (non-global) procedure to a procedure variable which isn't declared in the same block as the procedure being assigned. Such a variable might be called later, after exiting the scope in which the procedure was declared. The stack (local variables, etc.) assumed by the procedure will have been released, giving unpredictable operation, possibly fatal to the system.

Numeric Computation

Chapter 13

Introduction

When people think about computers, the first thing that they often think of is number-crunching, the giant calculator with a brain. Whether this is an accurate impression or not, numeric computations are an important part of computer programming.

Numeric computations deal exclusively with numeric values. Thus, adding two numbers or finding a sine or a logarithm are all numeric operations, while converting a number to a string or a string to a number are not. (Converting numbers to strings and strings to numbers is covered in the "String Manipulation" chapter.)

The most fundamental numeric operation is the assignment operation, achieved with the ":=" assignment operator. Thus the following statements are assignment statements:

```
A:=1;
Sine := sin(Theta);
X:=X+1;
```

Numeric Data Types

There are two numeric data types in Pascal, INTEGER and REAL. The valid range for REAL numbers is approximately:

$-1.797\,073\,134\,862\,315 \times 10^{308}$ through $1.797\,073\,134\,862\,315 \times 10^{308}$

The smallest non-zero REAL value allowed is approximately:

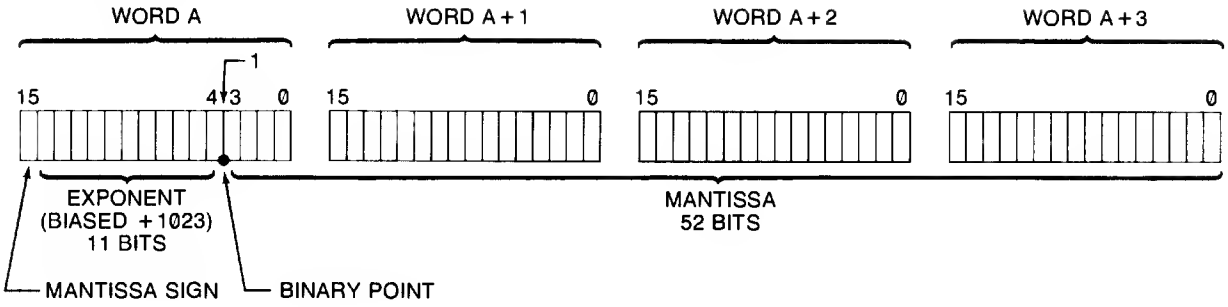
$\pm 2.225\,073\,858\,507\,202 \times 10^{-308}$

A REAL can also have the value of zero.

An INTEGER can have any whole-number value from:

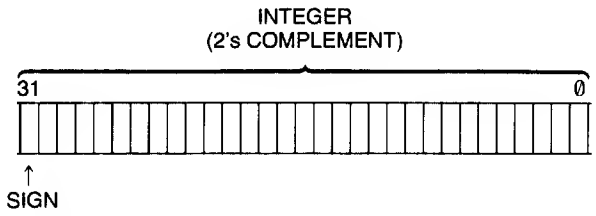
$-2\,147\,483\,648$ through $+2\,147\,483\,647$

Internal Numeric Formats



$$-1^{\text{mantissa sign}} \times 2^{\text{exponent} - 1023} \times 1.\text{mantissa}$$

Storage Format for REAL Variables



Storage Format for INTEGER Variables

Note

These formats are hardware dependent and operating system dependent. Other computers which support Pascal may have very different internal formats.

Declarations

In Pascal, you must declare all variables before using them, and both INTEGER and REAL data types are provided for declaring numeric variables:

```
var
  I, J:       integer;
  Days:       array [1..5] of integer;
  Weeks:      array [1..5] of array [1..17] of integer;
  X, Y:       real;
  Voltage:    array [1..4] of real;
  Hours:      array [1..5, 8..13] of real;
```

The above statements declare, both for integers and reals, the following:

- Two scalars,
- A one-dimensional array, and
- A two-dimensional array.

A scalar is a variable which can, at any given time, represent a single value. An array is a subscripted variable, and can contain multiple values, accessed by subscripts. You must specify both the lower and upper bounds of an array. Details on declarations of arrays and how to use them are provided in the "Data Types" chapter of this manual.

Type Conversions

The computer will automatically convert integers to real numbers in assignment statements and when parameters are passed by value in function and procedure calls. When parameters are passed by reference the conversion will not be made and a type-mismatch error will be reported. The computer will *not* automatically convert a real number to an integer; you must explicitly tell the computer to do it, and how to do it. There are two ways to convert a real number to an integer:

- The `round` function, which rounds the real value to the closest integer ($n.5$ rounds up to $n+1$),and
- The `trunc` function, which truncates the real value to the next integer toward 0.

For both of these functions, the sign (positive or negative) is not taken into account during the operation. You could think of them as doing these three operations:

1. Take the absolute value of the argument,
2. Do the operation,
3. Re-attach the original sign to the result.

An example of where this is significant is in the `trunc` function. It rounds toward 0, not toward $-\infty$. That is, `trunc(1.7)` yields 1, as expected, but `trunc(-1.7)` yields -1, not -2. It very literally truncates, it does not round to the next integer less than or equal to the argument.

Whenever numbers are converted from REAL to INTEGER representations, information can be lost. There are two potential problem areas in this conversion: rounding errors and range errors.

The computer may automatically convert between types when an assignment is made, and this presents no problem when an INTEGER is converted to a REAL. However, when you convert a REAL to an INTEGER, the REAL is modified (in whatever way you specified) to the closest INTEGER value. When this is done, all information about the value to the right of the radix (decimal point) is lost. If the fractional information is truly not needed, there is no problem, but converting back to a REAL will not reconstruct the lost information–it stays lost.

Another potential problem with REAL to INTEGER conversions is the difference in ranges. While REAL values range from approximately -10^{308} to $+10^{308}$, the INTEGER range is only from `minint` through `maxint`, or $-2\,147\,483\,648$ through $+2\,147\,483\,647$ (approximately -10^9 thru $+10^9$). Obviously, not all REAL values can be rounded into an equivalent INTEGER value. This problem can generate integer overflow errors.

While the rounding problem is important, it does not generate an execution error. The range problem *can* generate an execution error, and you should protect yourself from crashing the program by either testing values before assignments are made, or by using `try/recover` to trap the error[1], and making corrections after the fact.

1 See the chapter on "Error Trapping and Simulation" for details on error recovery.

The following fragment shows a method to protect against INTEGER overflow errors, although be aware that this method imposes minimum and maximum limits on the value[2]:

```
if X<minint then X:=minint
else
  if X>maxint then X:=maxint
  else
    X:=trunc {or round} (X);
```

Both these methods limit the excursion, but lose the fact that the values were originally out of range. If out-of-range is a meaningful condition, an error handling trap is more appropriate.

```
if (X<minint) or (X>maxint) then
  OutOfRange:=true
else
  X:=round {or trunc} (X);
```

Precision and Accuracy: The Machine Limits

Your computer stores all REAL variables with a sign, approximately 15 significant digits, and the exponent value. For most applications, this resolution is well beyond actual program needs. However, when high-resolution numerical analysis requires accuracy approaching the limits of the computer, round-off errors must be considered.

For many engineering and other applications, rounding errors are not a problem because the resolution of the computer is well beyond the limitations of most scientific measuring devices.

Rounding errors should be considered when conversions are made between decimal digits and binary form. Input/output operations are one time when this occurs. Given the format used for REALs, the conversion REAL→decimal→REAL will yield an identity only if the REAL→decimal conversion produces a 17-decimal-digit mantissa and the calculations for the conversions are done in extra precision. This is not the case on the Workstation Pascal. Therefore, several things can be said about these conversions on the Workstations:

- Up to and including 16 decimal digits are allowed when storing a number in internal form. If there are more digits, they are ignored.
- Up to and including 15 decimal digits may be output when converting a REAL for printing, display, etc. A full 16-digit conversion is not allowed because there are not 16 full digits of precision.
- It is possible for two distinct decimal numbers to map onto the same REAL number because the binary mantissa does not have enough bits to represent all 16 decimal digits. This can happen only if the decimal numbers are specified to 16-digits.
- It is possible for two distinct REAL numbers to convert to the same decimal number even if the conversion is done to 15-decimal-digit accuracy. Therefore, you cannot use a comparison of the digits in printed or displayed numbers to check for equality.
- All distinct 15 digit decimal strings have a correct distinct REAL representation, but it is not always possible to map them onto their correct representation because REAL multiplies are not done in extra precision, and the table entries are only 64 bits. In other words, the decimal→REAL conversion may produce a REAL that differs from the true representation by a maximum of two bits.

There are references at the end of this chapter to documents that contain further information on the subject of representing real numbers.

2 Later in this chapter, a method which truncates numbers outside the `minint..maxint` range is shown.

Evaluating Scalar Expressions

The Hierarchy

If you look at the expression 2+4/2+6, it can be interpreted several ways:

- 2+(4/2)+6 = 10
- (2+4)/2+6 = 9
- 2+4/(2+6) = 2.5
- (2+4)/(2+6) = .75

Computers do not deal well with ambiguity, so an arbitrary hierarchy is used for evaluating expressions to eliminate any questions about the meaning of an expression. When the computer encounters a mathematical expression, an expression evaluator is called. If you do not understand the expression evaluator, you can easily be surprised by the value returned for a given expression. In order to understand the expression evaluator, it is necessary to understand the valid elements in an expression and the evaluation hierarchy (the order of evaluation of the elements).

Six items can appear in a numeric expression; operators, constants, variables, intrinsic functions, user-defined functions and parenthesis. Operators modify other elements of the expression. Constants and variables represent numeric values in the system. Functions, both intrinsic and user-defined, return a value which replaces them in the evaluation of the expression. Parenthesis are used to modify the evaluation hierarchy.

The following table defines the hierarchy used by the computer in evaluating numeric expressions.

Math Hierarchy

Precedence	Operator
Highest	Parentheses; they may be used to force any order of operation
	Functions, both user-defined and machine-resident
	Multiplication and division: `*`, `/`, `mod`, `div`.
	Addition, subtraction, monadic plus and minus: `+`, `-`.
Lowest	Relational and Boolean operators: `=`, `<>`, `<`, `>`, `<=`, `>=`, `NOT`, `AND`, `OR`.

The boolean operators, `NOT`, `AND`, and `OR`, are included because of their utility in creating step functions (see the section "Step Functions" later in this chapter).

When an expression is being evaluated, it is read from left to right and operations are performed as encountered, unless a higher precedence operation is encountered immediately to the right of the operation encountered, or unless the hierarchy is modified by parentheses. If the computer cannot deal immediately with the operation, it is stacked, and the evaluator continues to read until it encounters an operation it can perform. It is easier to understand if you see how an expression is actually handled. The following expression is complex enough to demonstrate most of what goes on in expression evaluation.

```
A:=5+3*(4+2)/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0));
```

In order to evaluate this expression, it is necessary to have some historical data. Since trigonometric functions in Pascal deal only with radians, we will assume that `X` $= \pi/2$, and that the user-defined function `Neg1` returns -1. Evaluation proceeds as follows:

```
5+3*(4+2)/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
  5+3*6/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
    5+18/sin(X)+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
      5+18/1+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
       5+18+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
        23+X*ord(1>X)+Neg1*ord((X<1.5) and (X>0.0))
      23+X*ord(false)+Neg1*ord((X<1.5) and (X>0.0))
          23+X*0+Neg1*ord((X<1.5) and (X>0.0))
           23+0+Neg1*ord((X<1.5) and (X>0.0))
            23+Neg1*ord((X<1.5) and (X>0.0))
             23+-1*ord((X<1.5) and (X>0.0))
              23+-1*ord(false and (X>0.0))
               23+-1*ord(false and true)
                    23+-1*ord(false)
                       23+-1*0
                         23+0
                          23
```

The Delayed Binding Surprise

The computer delays binding of a variable to its value as long as possible. In the actual evaluation, a pointer to the location of a variable is what is stacked. This means that if a variable exists in an area of memory accessible to both the main program and a user-defined function, is used in an expression that also calls the user-defined function, and is modified in the function, the value of the expression can be surprising, although not unpredictable. For example, if we define a function `Negl` that returns a negative 1, we would expect the following lines to print 2.

```
X:=3;
Y:=X+Negl;
writeln(Y);
```

However, if the these lines are in the following environment:

```
program DBinding(input, output);
var
  X, Y:          integer;

function Negl: integer;
begin
X:=500;
Negl:=-1;
end;

begin
X:=3;
Y:=X+Negl;
writeln(Y);
end.
```

The actual result will be 499–surprising, but not unpredictable. The same thing will happen if the variable is passed by reference and modified in the user-defined function. Therefore, be careful when you use a user-defined function to modify values of global variables. They are designed for returning a single value, and are best reserved for that.

Operators

There are two types of numeric operators in Pascal: monadic and dyadic:

- A **monadic** operator performs its operation on the expression immediately to its right; e.g., `+`, `-`, `not`.
- A **dyadic** operator performs its operation on the two values it is between; e.g., `^`, `*`, `/`, `mod`, `div`, `+`, `-`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `and`, `or`.

  A **comparison** operator returns `true` or `false`, based on the result of a relational test of the operands it separates. The comparison operators are a subset of the dyadic operators that produce *boolean* results; e.g., `<`, `>`, `<=`, `>=`, `=`, `<>`.

While the use of most operators is obvious from the descriptions in the language reference, some of the operators have uses and side-effects that are not always apparent.

Expressions, Calls, and Functions

Numeric expressions can be passed by value to procedures and functions, if the corresponding formal parameter does not have the keyword `var` before the parameter name (assume for the moment it does not). Thus `5+X` is obviously passed by value. Not quite so obviously, `+X` is also passed by value. The monadic operator makes it an expression.

Step Functions

The comparison operators are obviously useful for conditional branching (`IF/THEN` statements), but are also valuable for creating numeric expressions representing step-functions. For example, let's try to represent the function:

- if `Select` < 0
  then `Result` $= 0$
- if $0 <=$ `Select` < 1
  then `Result` equals the square root of $A^2 + B^2$.
- if `Select` $>= 1$ (any other value)
  then `Result` $= 15$

It is possible to generate the required response through a series of `IF/THEN` statements, but it can also be done with the following expression:

```
Result:=0                    *ord(Select<0)+
        sqrt(sqr(A)+sqr(B))*ord((Select>=0) and (Select<1))+
        15                   *ord(Select>=1);
```

While the technique may not please the purist, it actually represents the step function very well. The boolean expressions cause the `ord` function to return a 1 or 0 which is then multiplied by the accompanying expression. Expressions not matching the selection return 0, and are not included in the result. The value assigned to `Select` before the expression is evaluated determines the computation placed in the result.

This technique can be used to represent a cyclicity as well; every time through a particular set of statements, the "next" of a list of variables is selected. At the end of the list, the list is repeated from the beginning, ad infinitum. Boolean expressions, constants, and variables can be included in numeric expressions if they are "converted" to numeric by the intrinsic function `ord`. What we haven't seen yet is that the boolean expressions can be generated by comparing *anything* that is comparable; e.g., numbers, characters, strings, etc. Note in the following examples, `X` and *a*, *b*, *c*, and *d* can be any of the type mentioned (numbers, characters, strings), as long as they are *all* of the same type. For example:

`X:=ord(X=0);` — This expression alternates the value of `X` between 0 and 1.

`X:=`*a*`*ord(X=`*b*`)+`*b*`*ord(X=`*a*`);` — This expression alternates between the arbitrary values of *a* and *b*, as long as $a \neq b$. Note that this algorithm can be extended to cycle through any number of values, as the next example shows.

```
X:=a*ord(X=d)+
   b*ord(X=a)+
   c*ord(X=b)+
   d*ord(X=c+
```

This expression cycles through the four values *a*, *b*, *c*, and *d*. Make sure $a \neq b \neq c \neq d \neq \ldots$, because if any value equals any other value, the process will loop without completing the series. As mentioned, this algorithm can be extended to any number of values, but it quickly gets cumbersome. The algorithm fills the need best when the values being cycled through exhibit no discernible pattern.

Another way of cycling through *n* unpatterned values is to make an array going from 1 to *n*, define the elements of the array to your numbers, and then cycle through the subscripts $1 \rightarrow n$.

Again: if, as in the above examples, `X` is a number, you could cycle through 14, 23, 4, −45, and 0. If, in the above examples, `X` is a character, you could cycle through `'x'`, `'&'`, `'A'`, and `'f'`. However, you cannot "multiply" a character by a number, but you can convert between characters and numbers easily. For example, assume `A`, `B`, `C`, `D` and `X` are of type `char`:

```
A:='x';
B:='&';
C:='A';
D:='f';
X:=A;          {or B or C or D...}
  . . .
X:=chr(ord(A)*ord(X=D)+
       ord(B)*ord(X=A)+
       ord(C)*ord(X=B)+
       ord(D)*ord(X=C));
```

If you want to cycle through strings, for example, 'Artichoke', 'I bought a centipede', 'quark', and 'Oh.', you could use the function strrpt for to do the "multiplication" (see the "String Manipulation" chapter for more on strrpt). For example (assume A, B, C, D and X are strings):

```
A:='Artichoke';
B:='I bought a centipede';
C:='quark';
D:='Oh.';
X:=A;          {or B or C or D...}
  . . .
X:=strrpt(A,ord(X=D))+
   strrpt(B,ord(X=A))+
   strrpt(C,ord(X=B))+
   strrpt(D,ord(X=C));
```

If you want to cycle through enumerated-type values, there is no easy way to do it other than putting the values in an array, and cycling through the subscripts.

Making Comparisons Work

If you are comparing integers, no special precautions are necessary. However, if you are comparing real values, especially those which are the results of calculations and functions, it is possible to run into problems due to rounding and other limits inherent in the system. For example, consider the use of comparison operators in if/then statements to check for equality in any situation resembling the following:

```
A:=2.53765477;
if sin(A)^2+cos(A)^2=1.0 then
  writeln('Equal')
else
  writeln('Not Equal');
```

You may find that the equality test fails due to rounding errors or other errors caused by the inherent limitations of finite machines. A repeating decimal or irrational number cannot be represented exactly in any finite machine.

A good example of equality error occurs when multiplying or dividing data values. A product of two non-integer values nearly always results in more digits beyond the decimal point than exists in either of the two numbers being multiplied. Any tests for equality must consider the *exact* variable value to its greatest resolution. If you cannot guarantee that all digits beyond the required resolution are zero, there are two techniques that can be used to eliminate equality errors:

- Use the absolute value of the difference between the two values, and test for the difference less than a specified limit. Here is an example of the absolute value method of testing equality. In this case, a difference of less than 0.001 is assumed to be evidence of adequate equality.

```
if abs(C-F)<0.001 then
  writeln('C is equal to F within 0.001')
else
  writeln('C is not equal to F within 0.001');
```

- Use the absolute value of the *relative* difference between two values, and test for the difference less than a specified limit:

```
if abs((C-F)/C)<10E-8 then
  writeln('Relative difference between C and F less than 10E-8')
else
  writeln('Relative difference between C and F greater than 10E-8');
```

This technique has the advantage that no additional statements are invested in overhead while preparing the data for evaluation. It also enables you to easily establish tolerance limits in making value comparisons, a capability that is useful in production and testing applications.

Numerical Functions

The resident functions are the functions that are part of the Pascal language (also called intrinsic). The following functions are available:

Function	Description
`abs`	Returns the absolute value of an expression.
`arctan`	Returns the arctangent of an expression.
`binary`	Returns the whole number value of a binary 32-bit integer. The argument is a string.
`cos`	Returns the cosine of the angle represented by the expression.
`exp`	Raise the Napierian *e* to a power. *e*≈2.718 281 828 459 05.
`hex`	Returns the whole number value of a hexadecimal 32-bit integer. The argument is a string.
`ln`	Returns the natural logarithm (Napierian base *e*) of an expression.
`octal`	Returns the whole number value of a octal 32-bit integer. The argument is a string.
`odd`	Takes an integer argument and returns a *boolean* result: `true` if and only if the integer argument is odd (the least significant bit is the opposite of the sign bit).
`round`	Returns the closest integer to the real argument. `round(X) = sign(X)*trunc(abs(X)+0.5)` where `sign(X) = ord(X>0.0) - ord(X<0.0)`. The result is of type `INTEGER`.
`sin`	Returns the sine of the angle represented by an expression.
`sqr`	Returns the square of an expression.
`sqrt`	Returns the square root of an expression.
`trunc`	Returns the integer part of the real argument; the fractional part is removed. The result is of type `INTEGER`.

Also, we should mention `div` and `mod`, although they are operators and not functions. The `div` operator does an integer divide; that is, it does a division and discards any fractional part. The `mod` operator returns the remainder of an integer division. `Z:=X mod Y` is equivalent to:

```
Z:=X-Y*(X div Y);
if Z<0 then Z:=Z+Y;
```

Dealing with Angles and Such

Before we get into the functions that deal with angles, let's discuss the angles themselves.

The Units

In Pascal, angles are always considered to be in radians, but people often want to deal with angles in degrees, or grads. Here are definitions of these units of angular measure:

- **Radians:** A radian is the unit of angular measure subtended by a piece of circumference of a circle whose length is equal to the radius of the circle. That is, if you take a line whose length is the radius of a circle, and bend that line around the circumference, the angle subtended by the bent radius is one radian.
- **Degrees:** A degree is 1/90 of a right angle. Thus, there are 360 degrees in a complete revolution.
- **Grads:** A grad is 1% of a right angle. Thus, there are 400 grads in a complete revolution.

In a nutshell, these are the relationships between these units of angular measure:

$$\begin{aligned}
\text{radians} &= \text{degrees} \times \tfrac{\pi}{180} &&\approx \text{degrees} \times 0.017\,453\,292\,519\,943\,3 \\
\text{radians} &= \text{grads} \times \tfrac{\pi}{200} &&\approx \text{grads} \times 0.015\,707\,963\,267\,949\,0 \\
\text{degrees} &= \text{radians} \times \tfrac{180}{\pi} &&\approx \text{radians} \times 57.295\,779\,513\,082\,3 \\
\text{degrees} &= \text{grads} \times \tfrac{9}{10} &&= \text{grads} \times 0.9 \\
\text{grads} &= \text{radians} \times \tfrac{200}{\pi} &&\approx \text{radians} \times 63.661\,977\,236\,758\,1 \\
\text{grads} &= \text{degrees} \times \tfrac{10}{9} &&\approx \text{degrees} \times 1.111\,111\,111\,111\,111\,1
\end{aligned}$$

The Functions

There are three functions which HP Pascal provides for dealing with functions: `sin`, `cos`, and `arctan`. However, since the default mode for all angular measure is radians, there needs to be a conversion from the units used in the program to radians. Assume, for example, that the program is considering all angles to be in degrees, and that the variable `Theta` holds the angle. The sine of `Theta` (which is in degrees) is:

```
Sine:=sin(Theta*0.01745329252);
```

When going the other direction, divide by the constant rather than multiply[3]. For example, say we want to calculate the arctangent of `X`, and have the answer in degrees:

```
Theta:=arctan(X)/0.01745329252;
```

Pascal traditionally is somewhat weak in the area of numerical functions, and HP Pascal is no different. A person can do hardly any trigonometric calculations from only the three functions provided, unless he knows some of the trigonometric relationships with which to derive the other needed functions. It is beyond the scope of this manual to provide very high speed algorithms to directly calculate the other trig functions; see a math book for those. However, it is within the scope of this manual to provide equations with which you can derive the appropriate functions from combinations of others. In the equations that follow, radians are assumed. If you wish to work in other units of angular measure apply the formulae above to convert to and from the desired unit of measure. Here are some of the trigonometric relationships.

3 Speed can be increased by multiplying by the reciprocal of the degree-to-radian number, rather than dividing by it. The computer does a multiply by 57.295 779 51 faster than a divide by 0.017 453 292 52.

Perhaps the best-known of the trig relationships is the following:

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

This shows that to calculate the tangent of an angle; take the sine of that angle and divide it by the cosine of that same angle. Note, however, that for odd multiples of $\pi/2$ (90°) the cosine is zero, so you must take appropriate measures to avoid dividing by zero in these cases.

A little less well-known are the arcsine and arccosine identities: The arcsine is defined thus:

$$\arcsin x = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

In Pascal:

```
Theta:=arctan(X/sqrt(1-sqr(X)));
```

Here also, there is some danger of dividing by zero. When $x = 1$, the denominator evaluates to zero. This divide by zero can be avoided in either of two ways. You could test for $x = 1$ and branch to a separate place to give the value, or a less cluttered way is the following:

$$\arcsin x = \arctan\left(\frac{x}{\sqrt{1-x^2+\epsilon}}\right)$$

where ϵ is some very small number e.g., 10^{-100}. Then, when $x = 1$, the denominator will not evaluate to zero, but a small number. The whole expression then evaluates to a very large (for all practical purposes, infinite) number. This is consistent with the desired behavior of the expression, because the argument for the arctangent function can range from $-\infty$ to $+\infty$.

Similar to the arcsine function is the arccosine function:

$$\arccos x = \arctan\left(\frac{\sqrt{1-x^2}}{x}\right)$$

But again, the divide-by-zero threat still exists; this time, when $x = 0$. The resolution of this problem is similar to that of the arcsine function above:

$$\arccos x = \arctan\left(\frac{\sqrt{1-x^2}}{x+\epsilon}\right)$$

or

```
Theta:=arctan(sqrt(1-sqr(X))/(X+Eps))
```

The reciprocals of the three main trig functions are the secant, cosecant, and cotangent:

$$\sec\theta = \frac{1}{\cos\theta}$$

$$\csc\theta = \frac{1}{\sin\theta}$$

$$\cot\theta = \frac{1}{\tan\theta}$$

This gives us the main three trigonometric functions `sin`, `cos`, and `tan`; their inverses `arcsin`, `arccos`, and `arctan`; and their reciprocals `sec`, `csc`, and `cot`.

The arctangent function supplied by the Pascal language is fine for many applications, but for others, it doesn't show you enough information. For example, say you have a point which has the Cartesian coordinates (3,4) and you want to determine its angle from the origin. This is quite simple: evaluate the arctangent of (Δy/Δx), or `arctan(4/3)`; it comes to about 0.93 radians, or about 53.13 degrees. This is the correct answer.

But here's the problem. Suppose that the point was $(-3,-4)$. Calculating arctan(Δy/Δx), or `arctan(-4/-3)` still results in 0.93 radians, or about 53 degrees, but this answer is off by 180 degrees! The problem arises from the fact that a negative number divided by a negative number gives the same answer as a positive number divided by a positive number.

Another problem arises when the point of interest lies on the Y-axis. This means x is zero, and again we're faced with a divide-by-zero problem.

The resolution of this requires that we pass both x and y to the new arctangent function; do not do the division beforehand, because then the signs are lost.

Let's look at every possibility of x and y. They both can be negative, zero, or positive.

$$\arctan(y,x) = \begin{cases} \arctan(y/x) + \pi & \text{if } x < 0 \text{ and } y < 0; \\ \arctan(y/x) + \pi \ (= \pi) & \text{if } x < 0 \text{ and } y = 0; \\ \arctan(y/x) + \pi & \text{if } x < 0 \text{ and } y > 0; \\ \frac{3}{2}\pi & \text{if } x = 0 \text{ and } y < 0; \\ 0 \text{ (by definition)} & \text{if } x = 0 \text{ and } y = 0; \\ \frac{1}{2}\pi & \text{if } x = 0 \text{ and } y > 0; \\ \arctan(y/x) & \text{if } x > 0 \text{ and } y < 0; \\ \arctan(y/x) \ (= 0) & \text{if } x > 0 \text{ and } y = 0; \\ \arctan(y/x) & \text{if } x > 0 \text{ and } y > 0; \end{cases}$$

As you can see, there is a pattern that emerges. If $x>0$, the normal arctangent function does well. If $x<0$, the normal arctangent function is consistently off by one-half revolution; we could just add π radians (180°) to the result. If $x = 0$, we need to check the sign of y and deal with it accordingly. If x and y are both zero, the arctangent is undefined; you're asking the computer to calculate the direction of a point. But, to keep the computer happy, let's define (somewhat arbitrarily) the result to be 0.

Taking the above information and translating it into Pascal, you come up with an arctangent function that goes something like the following (note that we're using one of the tricks we learned in the "Step Functions" section):

```
const
  Pi=            3.14159265358979323846;
     .
     .
     .
if X=0.0 then
  Atan:=(Pi/2
         +Pi*ord(Y<0.0))
            *ord(Y<>0.0)
else
  Atan:=arctan(Y/X)
        +Pi*ord(X<0.0)
        +2*Pi*ord((X>0.0) and (Y<0.0));
```

The "then" clause of the `if` statement says this:

1. We've already determined that $x = 0$; thus, the point is on the Y-axis. Therefore, assume $\pi/2$, or 90° (straight up).
2. Now, if $y<0$, add π (180°) to make the angle straight down.
3. One final check: if x and y are both zero, return zero.

The "else" clause of the `if` statement says this:

1. Take the arctangent of y/x.
2. If $x<0$, add π (180°).
3. If $x>0$ and $y<0$ (if the point is in the fourth quadrant), add 2π (360°). This ensures that the result ranges from 0 to 2π, rather than $-\pi/4$ to $+3\pi/4$.

Another class of trigonometric operations is the hyperbolic functions. Although we won't go into detail on how to use them, they are provided for your reference here.

$$\sinh z = \frac{e^z - e^{-z}}{2}$$

$$\sinh^{-1} z = \ln(z + \sqrt{z^2 + 1})$$

$$\operatorname{csch} z = \frac{1}{\sinh z}$$

$$\cosh z = \frac{e^z + e^{-z}}{2}$$

$$\cosh^{-1} z = \ln(z + \sqrt{z^2 - 1})$$

$$\operatorname{sech} z = \frac{1}{\cosh z}$$

$$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh^{-1} z = \frac{1}{2} \ln\left(\frac{1+z}{1-z}\right)$$

$$\coth z = \frac{1}{\tanh z}$$

Range Limits

It is sometimes necessary to limit the range of excursion of a variable (as in the discussion of REAL to INTEGER conversions mentioned in the introduction to this chapter). It is possible to do this with `if/then` statements:

```
if X>Maxx then X:=Maxx;
if X<Minx then X:=Minx;
```

It is more convenient to use `max` and `min` functions which can be defined thus:

```
function min(X, Y: real): real;
begin                                   {function "min"}
if X<Y then min:=X
else min:=Y;
end;                                    {function "min"}

function max(X, Y: real): real;
begin                                   {function "max"}
if X>Y then max:=X
else max:=Y;
end;                                    {function "max"}
```

For example:

```
X:=min(max(X,Minx),Maxx)
```

Note that `max` is used to establish the lower bound, and `min` is used to establish the upper bound. If you think about it a minute, it makes sense.

Truncating Real Numbers Outside the Integer Range

At the beginning of the chapter, a diagram was presented which showed the structure of the 64-bit REAL number in Workstation Pascal. In the 64 bits, there was a mantissa sign bit, an 11-bit exponent (biased by +1023), followed by 52 bits of mantissa. We will need to know this structure in order to implement the truncation algorithm.

What is necessary to truncate, at the decimal point, a real number is to zero out all the bits to the right of where the decimal point would be. These two steps are all that is involved:

- Determine the *binary* exponent. That is, look at the binary number represented by the 11 bits of exponent and subtract the 1023 bias. What is left is the exponent in base 2. This is how many bits of the mantissa to leave alone, because they represent digits to the left of the decimal point.
- Zero out all remaining bits of the mantissa. These bits being zeroed represent the digits to the right of the decimal point.

Implementing this algorithm is not quite so straightforward. As an example, let's walk through the algorithm, truncating 5.25. The exponent bits are `100 0000 0001` or 1025. Subtracting the bias of 1023, the actual exponent is found to be 2. The mantissa is the implicit "1.", followed by `0101`, followed by all zeroes, which evaluates to $1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$, or 1 + ¼ + 1/16. This must be multiplied by 2 taken to the power of the exponent; thus, the representation of 5.25 is 1.0101 (in binary) $\times 2^2$.

Since the exponent is two, we leave the first two bits of the mantissa alone, and zero out the rest. This results in $1\frac{1}{4} \times 2^2$, which is 5.

The following is a section of code which accomplishes the above task. As with all the code segments in this manual, it is written to be instructive, and not necessarily the most efficient spacewise or speedwise. Assume X is the variable being rounded.

There are two places where type coercion is necessary. Type coercion is the practice of interpreting the same set of bits in more than one way. In this case, we must take both a real value and an integer value and be able to set or clear individual bits within them.

```
type
  RealCoerce=    packed record
                   case integer of
                     1: (RealVal:     real);
                     2: (Bits:        packed array [1..64] of 0..1);
                 end;
  IntegerCoerce=packed record
                   case integer of
                     1: (IntVal:      -32768..32767);
                     2: (Bits:        packed array [1..16] of 0..1);
                 end;
```

A few variables are needed just for routine housekeeping:

```
var
   RCoerce:       RealCoerce;       {real variable which is interpreted 2 ways}
   ICoerce:       IntegerCoerce;    {16-bit integer which is interpreted 2 ways}
   Exponent:      integer;          {unbiased binary exponent}
   I:             integer;          {utility variable (loop counter) }
   Negative:      boolean;          {was the original number negative?}
```

And here are the guts of the procedure:

```
Negative:=(X<0.0);                   {remember if it was negative}
X:=abs(X);                           {let's just deal with positive numbers}
RCoerce.RealVal:=X;                  {put real value into coercion variable}
ICoerce.IntVal:=0;                   {initialize spare exponent variable}
for I:=2 to 12 do                    {for all exponent bits...}
  if RCoerce.Bits[I]=1 then          {...in the real value that are set...}
    ICoerce.Bits[I+4]:=1;            {...set a bit in the spare.}
Exponent:=ICoerce.IntVal-1023;       {get rid of bias}
if Exponent<0 then X:=0.0            {if X<1.0, 0.0 -> X}
else
  begin
  if Exponent<=52 then               {roundable?}
    for I:=Exponent+13 to 64 do      {all bits that should be cleared...}
      RCoerce.Bits[I]:=0;            {...are cleared.}
  X:=RCoerce.RealVal;                {change X}
  end;  {Exponent>=0}
if Negative then                     {was original number negative?}
  X:=-X;                             {    yep; put sign back}
```

Note

The above procedure is hardware dependent and operating system dependent and may not work on other Pascal systems.

Note that this algorithm truncates toward 0, as the Pascal `trunc` function does.

Rounding

Rounding occurs frequently in computer operations. The most common rounding occurs in printouts and displays, where it can be handled effectively with the formatting numbers (the numbers after the colons) in the output operation.

Sometimes it is necessary to round a number in a calculation, to eliminate unwanted resolution. There are three basic types of rounding:

- Rounding to a number of decimal places (limiting fractional information);
- Rounding up, down or to the nearest *x*, where *x* is any number, real, or integer, except zero; and
- Rounding to a total number of significant digits.

All three types of rounding have their own applications in programming.

Rounding to a Number of Decimal Places

The first, and most basic form of rounding is a special case of the first method above–that of rounding to a number of decimal places–but rounding always takes place to the nearest 10^0, or 1. The function to do this is called `round`, and it is part of the Pascal language. It was covered earlier in this chapter, with a reference to a subsequent function with which you could round real numbers *outside* the range of the integers. That function comes here.

In the previous section, an algorithm for truncating real numbers outside the range of `minint` to `maxint` was discussed. Using this same algorithm, rounding is only trivially more involved.

Rounding to the nearest integer (to the nearest 10^0) is merely a matter of truncating after adding 0.5. Imagine rounding 3.2 to the nearest integer; it rounds to 3. If we add 0.5 to 3.2, and then truncate, we again get 3. Imagine rounding 3.8 to the nearest integer; it rounds to 4. If we add 0.5 to 3.8, and then truncate, we get 4.

The only deviation from this algorithm is for negative numbers. What is often done is that the number is made positive, the rounding operation is done, and then the original sign is re-applied. For example, for rounding -3.2, you would note that it is negative, do the rounding operation on the absolute value of the argument, and re-apply the original sign to the result.

Note

In the rounding examples that follow, the range of numbers to be rounded is assumed to be in the `minint..maxint` range; thus, the standard Pascal function `round` will be used. If the numbers to be rounded are outside of this range, use the same algorithms stated, but do the rounding by adding 0.5 and then truncating with the "big number truncator" mentioned in the previous section.

The next logical step is to allow rounding to any power of ten, not just 10^0, as above. The idea is to eliminate decimal representation beyond a specific power of ten. A simple approach to it is to push the desired decimal information to the left of the radix, use `round` to get rid of the undesired decimal information, then reposition the radix correctly.

What must be done is this:

- Divide the number by the appropriate power of 10 to move the digit which will be the rightmost significant digit to just left of the decimal point.
- Round to 10^0, as usual.
- Multiply by the same appropriate power of ten to put the number back into the original order of magnitude.

For example, suppose you want to round 3.14159265 to the nearest 10^{-2}, or hundredth. First, you divide it by 10^{-2}, to get 314.159265. Next, round in the usual way, resulting in 314. Finally, multiply by 10^{-2}, to return the number to the original order of magnitude: 3.14.

Rounding to the Nearest X

All rounding applications don't fit nicely into the "power of 10" pattern mentioned above. What if you wanted to round to the nearest 25? Or 37? Or 0.123 or ¼? Some applications require rounding to the nearest multiple of some pretty unusual numbers.

This, again, is a logical extension of the previous method where we were dealing with powers of 10. Say, for example, that we want to round 19.2 to the nearest dozen. The method is simply:

```
Rounded:=round(19.2/12)*12;
```

Or, more generically, if `N` is the number to be rounded, and `M` is the number to be rounded to:

```
Rounded:=round(N/M)*M;
```

Rounding to N Significant Digits

There is a tendency for the number of decimal places to grow as calculations are performed on the results of other calculations. One of the first things covered in training for engineering and the sciences is how to handle the growth of the number of decimal places in a calculation. If the initial measurements from an experiment produced three digits of information per reading, it is very misleading to produce a seven-digit number as the result of a long series of calculations. Rounding to a specific number of significant digits allows you to eliminate the unwanted digits, to produce more realistic calculations and answers.

In the process of rounding to a certain number of significant digits, you must address than fact that you don't know where the decimal point is going to be, and the algorithm shouldn't care anyway. Taking this factor into account requires one more step than the previously mentioned rounding methods. The step is: Find out how far the decimal point has to be moved in order to position the number such that a regular rounding operation can be done. In all, the steps are as follows. Assume `X` is the number to be rounded, and `Digits` is the number of significant digits the result is to exhibit.

1. To find out how far the number is to be shifted, make a number which is the next larger order of magnitude; i.e., 1×10^n. This is accomplished by taking the logarithm[4] to the base 10, rounding it *up* to the next integer, and taking 10 to that power.
2. Shift the number again by dividing it by an appropriate power of 10 in order to take into account the number of digits to which to round.
3. Round in the usual way.
4. Shift back the number of digits found in Step 2.
5. Shift back the number of digits found in Step 1.

Implementing this in Pascal is not too difficult. Following is a section of code which would go into a function named `DRound`, which rounds to a certain number of digits. There are several ways to combine steps in this code segment to increase speed, but they were left out to maintain readability. Assume the functions `TenToThe` and `Log10` exist and calculate a power of ten, and the common log, respectively.

```
var
  DigitPower, MagnitudePower:   real;
      .
      .
      .
if Digits>=15 then
  DRound:=X
else
  if Digits<=0 then
    DRound:=X
  else
    begin
    MagnitudePower:=TenToThe(trunc(Log10(abs(X)))+1);
    DigitPower:=TenToThe(-Digits);
    X:=X/MagnitudePower;
    X:=round(X/DigitPower)*DigitPower;
    X:=X*MagnitudePower;
    DRound:=X;
    end;
```

4 Assume for the moment that functions exist whereby the common logarithm ($\log_{10}x$) common antilogarithm (10^x) can be obtained. The next section in this chapter illustrates how to implement these in Pascal.

Logarithms and Powers

There are two functions resident in Pascal which deal with logarithms: `ln`, which takes the natural log[5], and `exp`, which takes the natural antilog, or takes *e* to a power. With these two functions, we can do quite a bit.

X to the Yth Power

One of the logarithmic identities is

$$x^y = b^{y \log_b x}$$

where *b* is any non-zero number base. Since this works with any number base, *e* will work nicely:

$$x^y = e^{y \ln x}.$$

Knowing this, it is straightforward to take any number to any power. For example, to find out how many cubic feet in a cubic mile, you take 5280^3, or

```
CubicFeet:=exp(3*ln(5280));
```

When *x*, above, is a commonly used number, you can save computer time by calculating the natural log of it once, and than hard-coding it. The increase in speed can be significant, because both `exp` and `ln` are complex, relatively slow functions to calculate.

For example, to calculate 10 to a power, you could execute this statement every time:

```
Y:=exp(X*ln(10));
```

However, `ln(10)` is not going to change, so speed can be increased by converting this to an equivalent value: 2.302 585 092 994 05.

```
Y:=exp(X*2.30258509299405);
```

This approach can be taken with any number base.

The Xth Root of Y

Another logarithmic identity comes into play here:

$$\sqrt[x]{y} = y^{1/x}$$

This says that the *x* th root of *y* is obtained merely by taking *y* to the power of the reciprocal of *x*. After taking the reciprocal–just dividing the number into 1–use the approach immediately above for taking a number to a power.

In Pascal, to take the cube root of 27, it would be:

```
Y:=exp(1/3*ln(27));
```

5 The "natural logarithm" is a logarithm based on the Napierian number e, which equals approximately 2.718 281 828 459 045.

Log to Any Base

So far, we have been looking at logarithms to the base *e* exclusively, with a minor excursion into base 10. But logarithms exist in any base, so how can you figure a log to any user-specified base? The following derivation illustrates what can be done.

The definition of logarithms:	$\log_b x = y$
$\therefore$ (apply a logarithmic identity)	$b^y = x$
$\therefore$ (take natural log of both sides)	$\ln b^y = \ln x$
$\therefore$ (apply logarithmic identity to left side)	$y \ln b = \ln x$
$\therefore$ (divide both sides by $\ln b$)	$y = \ln x / \ln b$

What this means is that we can calculate the logarithm *to any base* of a number by dividing the log of the number by the log of the base. For example, to determine how many bits in a computer are required to represent a certain number–say 500–you need to take the log to the base 2, since bits deal in base 2.

```
Bits:=trunc(ln(500)/ln(2))+1;
```

Calendar Functions

A very useful capability for a computer to have is that of dealing with time. The Pascal operating system has some capabilities of dealing with time through the interface to the system clock. However, the clock is more designed to deal with centiseconds, seconds, minutes, and hours, than it is to deal with days, months, and years (although it can do some of this).

This section of the chapter deals with a broader area of timekeeping capabilities, ranging up to time spans of thousands of years.

The Julian Day

The Julian day, named in honor of Julius Caesar, is an astronomical convention representing the number of days that have elapsed since January 1, 4713 B.C. It is nothing more than an arbitrary "zero point" from which dates can be calculated. Since every month/day/year date has a Julian Day number, it becomes quite easy to determine how many days apart events are.

Converting Between Julian Day and Month/Day/Year

The formulae for determining the Julian Day number are these:

$$\text{Day}_{\text{Julian}} = \lfloor 365.25y' \rfloor - \lfloor y'/100 \rfloor + \lfloor y'/400 \rfloor + \lfloor 30.6001m' \rfloor + \text{day} + 1720997$$

where

$$y' = \begin{cases} \text{year} - 1 & \text{if month} \leq 2 \\ \text{year} & \text{if month} > 2 \end{cases}$$

$$m' = \begin{cases} \text{month} + 13 & \text{if month} \leq 2 \\ \text{month} + 1 & \text{if month} > 2 \end{cases}$$

This algorithm is valid only for dates after October 15, 1582, since a 10-day calendar correction was done at that time. If you include the 10-day correction, this October 15, 1582 limit no longer applies.

If an invalid date is sent to the routine, there will be no indication that the number coming back is wrong; you must check for out-or-range conditions yourself.

```
Yr:=Year-ord(Month<=2);
Mo:=Month+1+12*ord(Month<=2);
Julian:=trunc(365.25*Yr)-trunc(Yr/100)+trunc(Yr/400)+trunc(30.6001*Mo)+
 Day+1720997;
```

After converting a month/day/year into a Julian Day number and doing some desired operation, you'll need to convert a Julian Day number back into a month/day/year. These are the formulae you'll need:

$$d' = \text{Day}_{\text{Julian}} - 1720997$$

$$y' = \left\lfloor \frac{d' - 121.5}{365.2425} \right\rfloor$$

$$m' = \left\lfloor \frac{d' - \lfloor 365.25y' \rfloor + \lfloor y'/100 \rfloor - \lfloor y'/400 \rfloor}{30.6001} \right\rfloor$$

$$\text{day} = d' - \lfloor 365.25y' \rfloor + \lfloor y'/100 \rfloor - \lfloor y'/400 \rfloor - \lfloor 30.6001m' \rfloor$$

$$\text{month} = \begin{cases} m' - 13 & \text{if } m' \geq 14 \\ m' - 1 & \text{if } m' < 14 \end{cases}$$

$$\text{year} = \begin{cases} y' & \text{if month} > 2 \\ y' + 1 & \text{if month} \leq 2 \end{cases}$$

In Pascal:

```
D:=Julian-1720997;
Y:=trunc((D-121.5)/365.2425);
Temp:=D-trunc(365.25*Y)+trunc(Y/100)-trunc(Y/400);
M:=trunc(Temp/30.6001);
Day:=Temp-trunc(30.6001*M);
Temp:=M-1-12*ord(M>=14);
Month:=Temp;
Year:=Y+ord(Temp<=2);
```

These two functions allow you to do many desirable things (assume you have a function `Julian` which calculates the Julian day number, and a function `mmddyyyy` which calculates month/day/year):

- *How many days apart were Event A and Event B?*

  `DaysApart:=Julian(`⟨*Event B date*⟩`)-Julian(`⟨*Event A date*⟩`);`

- *What day of the year is June 18, 1985?*

  `DayOfYear:=Julian(`⟨*June 18, 1985*⟩`)-Julian(`⟨*January 1, 1985*⟩`)+1;`

- *What will be the date 200 days from today?*

  `Date:=Mmddyyyy(Julian(`⟨*today*⟩`)+200);`

Day of the Week

The Julian Day number also lends itself nicely to finding out which day of the week on which a particular date fell.

```
DayOfWeek:=(Julian(Month, Day, Year)+1) mod 7+1;
```

This algorithm returns a number in the range 1 – 7, meaning Sunday – Saturday, respectively.

Leap Year

As mentioned, a leap year is a year in which an extra day is placed at the end of February. The current algorithm, instituted along with the Gregorian Calendar, is this: A year is not a leap year unless it is a multiple of 4, in which case it is, unless it is a multiple of 100, in which case it is not, unless it is a multiple of 400, in which case it is[6].

In Pascal, this is:

```
if Year mod 4<>0 then
  LeapYear:=false
else
  if Year mod 100<>0 then
    LeapYear:=true
  else
    if Year mod 400<>0 then
      LeapYear:=false
    else
      LeapYear:=true;
```

6 Got that?

Number Base Conversion

Utility functions are available with the Pascal language to simplify some of the conversions between number bases. The three functions `binary`, `octal`, and `hex` convert strings representing numbers in base 2, 8, and 16, respectively, to integers. There are no standard Pascal functions which convert integers to these bases.

For those applications where you must deal with number bases other than 2, 8, or 16, you must create your own conversion routines.

To refresh your memory on conversion of a number in another base to base 10, consider the following. You want to convert 1432 in base 5 to base 10. This is $1\times5^3+4\times5^2+3\times5^1+2\times5^0$, or $125+100+15+2$, or 242.

To convert 242 back to base 5, you take successive powers of 5 until the first time a power of five is greater than the original number, then back off one, and this is where you start. For example:

$5^0=1$	$1<242$, so increment the exponent.
$5^1=5$	$5<242$, so increment the exponent.
$5^2=25$	$25<242$, so increment the exponent.
$5^3=125$	$125<242$, so increment the exponent.
$5^4=625$	$625>242$, so decrement the exponent; we've found where to start.

Thus, the power of 3 is where we are to start subtracting:

How many 5^3s can be taken from 242?	One; write `1`, do the subtraction.
How many 5^2s can be taken from 117?	Four; write `4`, do the subtraction.
How many 5^1s can be taken from 17?	Three; write `3`, do the subtraction.
How many 5^0s can be taken from 2?	Two; write `2`, do the subtraction.

At this point, the iterations stop, because the original number has been reduced to zero. We've successfully converted 242 in base 10 to base 5; we've written `1432`, which is the original number.

The following code segments illustrate what must be done to convert between base 10 and base *n*. Note that the numbers which are in base 10 are regular integers, and the numbers in other bases are represented as strings, because any base greater than 10 requires letters for the other digits.

Here is an algorithm for converting an integer (N) into a string (Strng) representing a number in base *n* (Base).

```
const
  Chars=                '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'; {base 36 max}
type
  Str32=                string[32];
var
  Power:                integer;
  StrIndex,CharsIndex:  integer;
  Strng:                Str32;
      .
      .
      .
Power:=1;                                 { \   Find out what     }
repeat                                    {  \  number to start   }
  Power:=Power*Base;                      {   > dividing the      }
until Power>N;                            {  /  input parameter   }
Power:=Power div Base;                    { /   by.               }
Strng:=strrpt(' ',32);                    {initialize the result string}
StrIndex:=0;                              {where are we in the string?}
repeat
  CharsIndex:=N div Power;                {get magnitude of "digit" in base n}
  StrIndex:=StrIndex+1;                   {increment character pointer}
  Strng[StrIndex]:=Chars[CharsIndex+1];   {place "digit" in appropriate position}
  N:=N mod Power;                         {subtract digit*Power from N}
  Power:=Power div Base;                  {decrement exponent}
until Power=0;                            {until number goes to 0}
```

Here is an algorithm for converting a string representing an number in base *n* to an integer:

```
const
  Zero=                 ord('0');
var
  l, Pos, Temp:         integer;
  StrChar:              string[1];
  BadChar:              boolean;
      .
      .
      .
if (Base<2) or (Base>36) then
  begin
  writeln('Error: Base=',Base:0);
  halt(-1);
  end;
BadChar:=false;
for l:=1 to strlen(Strng) do
  begin
  StrChar:=str(Strng,l,1);
  Pos:=strpos(Chars,StrChar);
  if (Pos<1) or (Pos>Base) then
    BadChar:=true;
  end;
if BadChar then
  ⟨error message⟩
else
  begin
  Temp:=0;
  for l:=1 to strlen(Strng) do
    begin
    StrChar:=str(Strng,l,1);
    Pos:=strpos(Chars,StrChar);
    Temp:=Temp*Base+Pos-1;
    end;
  ⟨function name⟩:=Temp;
  end;
```

Random Numbers

In many mathematical and statistical fields of study, there is a need to simulate random events. A random event is an event which does not produce the same outcome every time it occurs under identical circumstances. And, since many events and processes can be mathematically modeled, a computer should be able to model random events.

Technically, a computer is hard-pressed to generate real random numbers, because the one of the requirements of a sequence of random numbers is that the value of any particular number is completely unrelated to its previous and succeeding neighbor. Most computerized random number generators generate random numbers *algorithmically*, that is, the value of each number is somehow derived from the previous one (or *n* numbers).

Since the definition of "random number sequence" requires that neighboring numbers be unrelated, algorithmic random number generators do not *really* generate random numbers. On the other hand, the sequence of numbers generated by a good algorithmic random number generator passes batteries of randomness tests, therefore the numbers can be considered random. To remove the apparent paradox here–random or not random–computer scientists have called the numbers generated by algorithmic random number generators "pseudo-random" numbers.

Workstation Support of Pseudo-Random Numbers

Your computer has two routines which deal with random numbers. Both of them are exported from module `rnd` (in `SYSVOL:LIBRARY`):

`random` This procedure generates a random number "seed." The seed of a pseudo-random number generator is a number from which the next value in a sequence of pseudo-random numbers is generated. Typically, this routine is called once per sequence of random numbers. Its declaration is:

```
procedure random(var seed: integer);
```

The random-number seed "`Seed`" must be initialized prior to use. A good initial value for `Seed` is one with several digits, where the least significant digit is a 1, 3, or a 7.

`rand` This function takes the pseudo-random number seed generated by `random` and returns a pseudo-random integer in a user-definable range, as well as updating the seed for the next iteration. This routine is called every time a pseudo-random number is needed; not just once per sequence, as `random` was. Its declaration is:

```
function rand(var seed: integer; range: shortint): shortint;
```

The type `shortint` indicates a signed, two's complement, 16-bit integer. It is exported from the `sysglobals` module (in `CONFIG:INTERFACE`):

```
type
  shortint=          -32768..32767;
```

The parameter called `range` allows you to specify the integer range within which the returned pseudo-random integer will be. That is, if you invoke this function with range equal to *n*, the returned integer will be in the range 0 through $n-1$, inclusive. Obviously, you can add 1 to the function result if you wish the range to be 1 through *n*.

Note that the parameter `Seed` will be changed by a call to either `random` or `rand`.

Using the Pseudo-Random Number Generator

Generating a sequence of pseudo-random numbers between 0 and $n-1$ or between 1 and n is trivial, if repetition–having the same number more than once–is permitted. It is as easy as:

```
I:=rand(Seed,⟨n⟩);      {Range: 0..⟨n-1⟩, inclusive}
I:=rand(Seed,⟨n⟩)+1;    {Range: 1..⟨n⟩, inclusive}
```

If an arbitrary set of limits is desired, say, you want pseudo-random integers between m and n ($m \leq n$), this is as easy as:

```
I:=rand(Seed,⟨n⟩-⟨m⟩+1)+⟨m⟩;          {Range: ⟨m⟩..⟨n⟩, inclusive}
```

The following program does just that: it generates 100 pseudo-random integers in a user-defined range:

```
program Randoms(input,output);
import rnd;                                {get the random number routines}
var
  Seed, Rmin, Rmax, I:  integer;
begin
Seed:=12345;                               {initialize the random number seed}
write('Range for random numbers: ');       {ask for...}
readln(Rmin,Rmax);                         {...and receive the range limits.}
for I:=1 to 100 do                         {100 times...}
  write(rand(Seed,Rmax-Rmin+1)+Rmin:0,' '); {...write a number between...}
                                           {...Rmin and Rmax.}
end.
```

If repetition is not allowed, it is not quite as straightforward, although it is not difficult.

An example of generating a random sequence without repetition is shuffling a deck of cards. No matter how poorly or how well the randomness is applied, there will never be more than one ace of spades[7], or seven of clubs, etc.

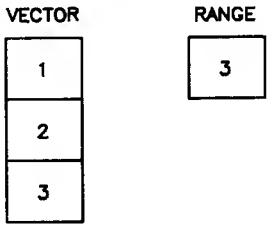
The following example concerns generating a pseudo-random number sequence of arbitrary size *without* repetition. The routine in this example generates n pseudo-random integers between 1 and n, although it could easily be modified to generate fewer than n integers in the range 1 to n, or to generate integers between m and n.

7 We are dealing with a regular deck of cards here, not a pinochle deck. Our deck has Ace through King of each suit.

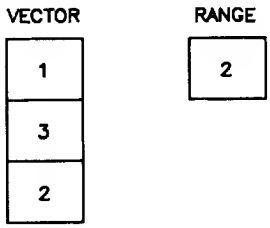
A Shuffling Algorithm

Generating a list of non-repeating pseudo-random numbers is not difficult. Let's go through the algorithm by hand to generate a list of 3 pseudo-random numbers. There needs to be a vector–a one-dimensional array–3 elements long, through which the shuffled integers will be returned to the calling routine. In addition to this, we need a temporary storage area of type `integer`. There also needs to be a variable called `Range` which specifies the maximum value the random number can be when selecting elements from the array.

1. Define `Vector[I]:=I`. The setup looks like this:

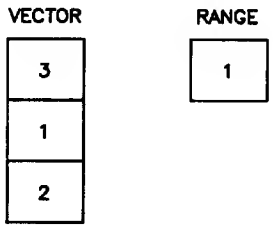


2. Pick a pseudo-random number, ⟨*rnd*⟩, in the range 1 through `Range`, which is currently 3. Let's say ⟨*rnd*⟩ is 2.
3. Switch the values of `Vector[Range]` and `Vector[`⟨*rnd*⟩`]`. We have now defined the pseudo-random number in the element of `Vector` specified by `Range` (now 3). Decrement `Range`. The setup now looks like this:



4. Pick a pseudo-random number in the range 1 through `Range`, which is currently 2. Let's say ⟨*rnd*⟩ is 1.

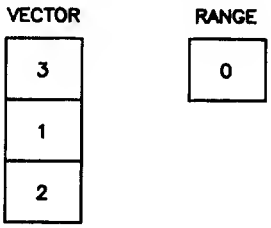
5. Switch the values of `Vector[Range]` and `Vector[`⟨*rnd*⟩`]`. We have now defined the pseudo-random number in the element of `Vector` specified by `Range` (now 2). Decrement `Range`. The setup now looks like this:



6. The final step of this algorithm is virtually a no-op. It is driven by "Pick an integer between 1 and 1, inclusive." However, for the sake of completeness, we will go through it.

   Pick a pseudo-random integer in the range 1 through `Range`, which is currently 1. Obviously, ⟨*rnd*⟩ must have the value of 1.

7. The switching of the values in `Vector[Range]` and `Vector[`⟨*rnd*⟩`]` doesn't change anything this time. Decrement `Range`. The setup looks like this:



8. `Range` has been reduced to 0, so we are done. Return the array to the calling routine.

The Shuffling Routine

Putting the above algorithm into Pascal is quite simple, as the following example shows:

```
$SYSPROG$
PROGRAM Shuffle_(OUTPUT);
IMPORT rnd;                              {get RANDOM and RAND}
VAR
  Vector1:      ARRAY [1..10] OF INTEGER;
  Vector2:      ARRAY [1..20] OF INTEGER;
  I:            INTEGER;
$PAGE$  {*************************************************************************}
PROCEDURE Shuffle(VAR Vector: ARRAY [Lo..Hi: INTEGER] OF INTEGER);
VAR
  Temp:         INTEGER;               {temporary storage area}
  Seed:         INTEGER;               {pseudo-random number seed}
  Range:        INTEGER;               {maximum random number}
  I, J:         INTEGER;
BEGIN
Seed:=1234567;                         {initialize the random number seed}
FOR I:=Lo TO Hi DO                     {initialize the temporary array}
  Vector[I]:=I;
Range:=Hi;                             {pick from whole thing the first time}
FOR I:=Lo TO Hi DO
  BEGIN
  J:=rand(Seed,Range)+1;               {where does next element go?}
  Temp:=Vector[Range];                 { \  Switch locations  }
  Vector[Range]:=Vector[J];            {  > of Vector[Range]  }
  Vector[J]:=Temp;                     { /  and Vector[J].    }
  Range:=Range-1;                      {reduce the choice range}
  END;
END;
$PAGE$  {*************************************************************************}
BEGIN
Shuffle(Vector1);
WRITE('Shuffled vector: '#13#10'  ');
FOR I:=1 TO 10 DO WRITE(Vector1[I]:0,STRRPT(', ',ORD(I<10)));
WRITELN;
WRITELN;
Shuffle(Vector2);
WRITE('Shuffled vector: '#13#10'  ');
FOR I:=1 TO 20 DO WRITE(Vector2[I]:0,STRRPT(', ',ORD(I<20)));
WRITELN;
END.
```

There are several features of note in the above example:

- The array to be filled with non-repetitive pseudo-random numbers is passed to the shuffling routine as a conformant array. The routine automatically adjusts its behavior to deal with whatever size array was passed to it. Note that since the array is passed as a conformant array, it may not be portable to other Pascal systems. (See the chapter "Data Structures" for more information on conformant arrays.)
- Using control characters (carriage-return and line-feed) in a string being printed. (See the chapter "String Manipulation" for more information on strings.)
- Using the step-function capability provided by using `ord(`⟨*boolean value*⟩`)` as a number. (See the section "Step Functions" in this chapter for more information on these.)

The following are references which contain further information on numeric computation.

Coonen, Jerome T.; "An Implementation Guide to the Proposed Floating Point Standard", *Computer Magazine*, Jan. 1980.

Cody, William J. Jr. and William Waite; *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

Sterbenz, Pat H.; *Floating Point Computation*, Prentice Hall, 1974.

Signum Newsletter, Oct 1979.

Notes

Chapter 14 String Manipulation

Introduction

It is often desirable to store non-numerical information in the computer. A word, a name or a message can be stored in the computer as a **string**. Any sequence of characters, both displayable and non-displayable, may be used in a string. Apostrophes (') , or single quote marks, are used to delimit the beginning and end of the string. The following are valid string variable assignments.

```
A:='COMPUTER';
Fail:='The test has failed.'#7;  {the "#7" is a CTRL-G (bell) }
File_name:='INVENTORY';
TEST:=str(Fail,5,4);
```

The variable name (the left-hand side of the assignment) is assigned to the string value specified by the right-hand side of the assignment.

The *length* of a string is the number of characters in the string. In the previous example, the length of `A` is 8 since there are eight characters in the literal `'COMPUTER'`; you don't count the quotes, since they are only used to delimit the beginning and end of the string value.

Pascal allows the dimensioned length of a string to range from 1 to 255 characters. The current length (number of characters in the string) may range from zero to the dimensioned length. A string of zero characters is called a "null string" or an "empty string". An error results whenever you try to assign a string variable more characters than its dimensioned length.

Special Cases of String Assignment

A string may contain any character. There are three special cases:

- The quote mark itself,
- Control characters (`ord`<32), and
- The upper half of the character set (`ord`>=128).

Getting a Quote Into the String

To get the quote mark (or "apostrophe") itself into the string requires two quotes in succession:

```
Quoted:='The time is ''NOW'',';
Apostrophe:='''';
writeln(Quoted);
writeln(Apostrophe);
```

Produces:

```
The time is 'NOW',
'
```

Getting a Control Character Into a String

To get control characters whose ordinal value is less than 32 into a string, you put a character or an integer[1] after a pound sign (a "#"). Say that you wanted a string to contain an "A", a carriage return, and a "B". You could type:

```
Strng:='A'#M'B';
```

The pound sign and the character following are converted into `chr(ord(`⟨*character*⟩ `mod 32)`. An ASCII table will provide information on what values to use.

Note that these characters cannot be *inside* quote marks, or you will end up with just those characters. For example, if the two inner quote marks in the above example were removed, the string would consist of an "A", a "#", an "M", and a "B".

In the same way as a non-numeric character can follow a pound sign, a number can, too. To get the same string as the above example, you could type:

```
Strng:='A'#13'B';
```

Again, notice that the pound sign and its number must be outside of quotes.

Getting "Other" Characters Into a String

The "pound-sign-character" method mentioned above is limited to creating characters whose `ord` is less than 32. The "pound-sign-integer" method has no such restriction; it can create any character between `chr(0)` and `chr(255)`, inclusive.

For example, if your machine supports underlining text, you can cause a string to contain its own underline on/off characters:

```
Strng:=#132'This is underlined,'#128;
```

1 If you put an integer after the #, you are not limited to characters whose `ord` is less than 32. See the next section.

Declaring String Variables

The following statements may be used to declare a string:

```
type
  Str20=              string[20];
var
  MyString:           Str20;
```

or

```
var
  MyString:           string[20];
```

String Length

A string may be declared (dimensioned) to any length between 1 and 255 characters, inclusive. The `var` statement declares and reserves storage for string variables.

```
const
  ShortStringLength= 4;
type
  ShortStringType=   string[ShortStringLength];
var
  ShortString:       ShortStringType;
  LongString:        string[255];
```

Strings that have been allocated but not assigned can contain anything; there is no automatic housekeeping done. Therefore, we highly recommend initializing string variables to some known state before use.

String Storage in Memory

Strings, as all other Pascal variables, must have space reserved before assignment. That space reserved consists of one length byte, followed by as many characters as specified in the declaration (the length byte is a one-byte area at the beginning of every string which indicates, in its eight bits, the current length of the string). The storage area is aligned along an even-byte boundary. Thus, a variable declared as `string[6]` will consume 8 bytes: the six bytes desired, the length byte, and another byte for padding to an even-byte boundary.

String Arrays

Strings, like any other data type in Pascal, can be incorporated in arrays and records. Large amounts of text are easily handled in arrays. For example:

```
var
  BigArray:           array[1..1000] of string[80];
```

This statement reserves storage for 1000 lines of 80 characters per line. Each string in the array can be accessed by an index. For example:

```
writeln(BigArray[27]);
```

Prints the 27th element in the array.

Since each character in a string uses one byte of memory and each string in the array requires as many bytes as the length of the string (plus one, for the current length, plus possibly another one for the even-byte-boundary pad character), string arrays can quickly use a lot of memory.

As an example of using a string array, a source program saved on a disc file can be entered into a string array, manipulated, and written back out to disc.

Evaluating Expressions Containing Strings

Evaluation Hierarchy

Evaluation of string expressions is simpler than evaluation of numerical expressions. The two allowed operations are concatenation and parenthesization. The evaluation hierarchy is presented in the following table.

Order	Operation
High	Parentheses (functions, which require parenthesized parameters, are included here).
Low	Concatenation

String Concatenation

Two separate strings are joined together by using the concatenation operator "+". The following program segment combines two strings into one.

```
One:='WRIST';
Two:='WATCH';
Concat:=One+Two;
writeln(One,' ',Two,' ',Concat);
```

Prints:

```
WRIST WATCH WRISTWATCH
```

The concatenation operation, in the third line, appends the second string to the end of the first string. The result is assigned to a third string. An error results if the concatenation operation produces a string value that is longer than the dimensioned length of the string variable to which it is being assigned.

To increase the readability of certain programs, parentheses can be used to force concatenation in a particular order. Note that the outcome result will be the same with or without parentheses, since all string operators (there is only the one) have the same priority. This is different from numeric expression evaluation, where there are several different operations, having different priorities.

```
CombinedString:=Strng1+(Strng2+Strng3);
```

Relational Operations

The relational operators used for numeric expression evaluation can also be used for the evaluation of strings. Testing begins with the first character in the string and proceeds, character by character, until the relationship has been determined.

The following examples show some of the possible tests.

```
'ABC' = 'ABC'                    True
'ABC' = ' ABC'                   False

'ABC' < 'AbC'                    True
'6'   > '7'                      False
'60'  > '7'                      False

'long' <= 'longer'               True
'RE-SAVE' >= 'RESAVE'            False
```

Any of these relational operators may be used: <, >, <=, >=, =, <>.

The outcome of a relational test is based first on the characters in the strings and, second, on the length of the strings. For example:

```
'BRONTOSAURUS' < 'CAT'
```

This relationship is true since the letter "C" is higher in ASCII (or *ordinal*) value than the letter "B". However, in the following example, the string length *is* taken into account:

```
'HIPPO' < 'HIPPOPOTAMUS'
```

In this case, all the characters match up through the point at which one string ends. At this point, the shorter string is considered the lesser.

String Functions

Several intrinsic functions are available in HP Pascal for the manipulation of strings. These functions include:

- Extracting substrings
- Determining string length and maximum string length
- Locating substrings within strings
- Conversion between string and numeric values
- Conversion between strings and packed arrays of characters
- Trimming off leading and/or trailing blanks
- Repeating strings zero or more times

Substrings

Using the string function `str`, you can extract a portion of a string, called a *substring*, from the source string. A substring may comprise all or just part of the original string. The `str` function requires three parameters:

- The source string expression
- The starting index of the substring
- The substring length

For example, assuming `Strng` is a string variable dimensioned to a maximum length of 20, and that it currently has the 16-character value of `'abcdefghijklmnop'`:

`str(Strng,3,4)`	specifies a substring of `Strng` starting at the third character and extending for 4 characters: `'cdef'`.
`str(Strng,16,1)`	specifies a substring of `Strng` starting at the sixteenth character and extending for 1 character: `'p'`.
`str(Strng,3,0)`	specifies a substring of `Strng` starting at the third character and extending for zero characters: `''`.
`str(Strng,39,4)`	specifies a substring of `Strng` starting at the thirty-ninth character and extending for 4 characters: **Error!**
`str(Strng,60,0)`	specifies a substring of `Strng` starting at the sixtieth character and extending for zero characters: *No* error.

Except for null substrings, the integer expression specifying the starting position of the substring must be in the range 1 to the current length of the string.

Substrings may appear only on the right side of an assignment statement.

Current Length of a String

The "length" of a string is the number of characters in the string. The `strlen` function returns an integer whose value is equal to the string length. The range is from 0 (given by the null string) through 255. For example:

```
write(strlen('HELP ME'));
Strng:='Greetings!';
writeln(strlen(Strng));
```

Prints: 7 10

Maximum Length of a String

This function returns the maximum length a string can legally be. This is its length as specified in its declaration, e.g., `string[80]`.

The `strmax` function can be used to avoid run-time errors which would occur from string overflows. For example:

```
if strlen(Strng)+strlen(Addendum)>strmax(Strng) then
  writeln('String would overflow.  Append operation not performed.')
else
  Strng:=Strng+Addendum;
```

Substring Position

The "position" of a substring within a string is determined by the `strpos` function. The function returns the value of the starting position of the *first occurrence of the substring* or zero if the entire substring was not found. For instance:

```
writeln(strpos('APPEAR','DISAPPEARANCE'));
```

prints `4`, because the string `'APPEAR'` is found in the string `'DISAPPEARANCE'`, and it starts in fourth character position.

The compiler option `$switch_strpos$` reverses the interpretation of the arguments in a `strpos` call. This brings the order of arguments into agreement with the HP Pascal Standard (which is also in agreement with the HP BASIC definition of `POS`). That is, if `$switch_strpos$` is in effect, the above example would have been coded:

```
writeln(strpos('DISAPPEARANCE','APPEAR'));
```

If `strpos` returns a non-zero value, the entire substring occurs in the first string and the value specifies the starting position of the substring.

The `$switch_strpos$` directive, if it is used, must appear at the beginning of the program. It sets (it doesn't complement) an internal flag which specifies that the interpretation order of `strpos` parameters should conform to the HP standard; thus, multiple occurrences of `$switch_strpos$` do not keep toggling the interpretation order.

Sometimes, you may not care *where* a substring is in a string, you need to find out only *if* it is in the string. Again, the `strpos` function avails itself:

```
$switch_strpos$
      .
      .
      .
var
  MasterList:          string[255];
  Item:                string[10];
  Found:               boolean;
      .
      .
      .
Found:=(strpos(MasterList,Item)>0);
if Found then ...
```

Note that `strpos` returns only the *first* occurrence of a substring within a string. By extracting a substring, and indexing through it, the `strpos` function can be used to find any occurrence or all occurrences of a substring. The following algorithm uses this technique to find any specified substring from a source string.

Assume that the source string–that string to be searched–is called `Source`, and that the substring you are looking for is called `Pattern`. Further, assume that the occurrence of the substring you are looking for is an integer called `Occurrence`. In other words, if you are looking for the third occurrence of "`is`" in the string "`This is the Mississippi`", you would set `Source` to "`This is the Mississippi`", `Pattern` to "`is`", and `Occurrence` to 3.

Note that in this algorithm, we are not permitting overlapping occurrences of the pattern sought. Thus, there is only one occurrence of "issi" in "Mississippi"; it starts in character 2. The occurrence starting at character 5 is not considered because the search resumes at character 6.

The following steps are required:

1. Find, in the whole of `Source`, the position of the first occurrence of `Pattern`. Place this value in the integer `Pos`.
2. If `Pattern` exists in the section of `Source` scanned (and if we haven't found the one we're looking for yet), do the following:
   a. Make note of the fact that you've found an(other) occurrence of `Pattern`.
   b. If we've found the one we're looking for, return the location of `Pattern` within the section of `Source` we just searched. If we haven't found the one we're looking for, search `Source` *from the first point* another occurrence of `Pattern` could exist, and, if it exists, note its position in `Pos`. What is meant by "from the first point another occurrence could exist" is this: the second occurrence of a string cannot occur (by our rules) until *after* the first occurrence ends. Thus, skip over the part of the string occupied by the characters before `Pos`, as well as the entire length of the first occurrence of the pattern.
   c. Go to Step 2.
3. We got out of the Step 2 loop because either (1) no more occurrences were found, or (2) we found the occurrence we were looking for. If we found the one we're looking for, return the location of `Pattern` within the whole of `Source`. If we didn't find the one we're looking for, return zero; the specified occurrence does not exist.

In Pascal, the following code segment accomplishes the desired task. Assume that this is part of a string function whose declaration looks like this:

```
function strpos2(Source, Pattern: Str255; Occurrence: integer): Str255;
```

where `Str255` is a type specifying `string[255]`.

```
if Occurrence=1 then                    {if looking for the first one...}
  strpos2:=strpos(Source,Pattern)           {...use the system routine.}
else
  begin                                 {otherwise...}
  Start:=1;                             {where to start search in Source}
  Found:=0;                             {how many have we found?}
  Plength:=strlen(Pattern);             {length of pattern searched for}
  Done:=false;                          {done yet?
  Pos:=strpos(Source,Pattern);          {search for Pattern in Source}
  while (Pos>0) and not Done do         {if we're still going...
    begin
    Found:=Found+1;                     {eureka! another one!}
    if Found=Occurrence then            {the one we're looking for?}
      Done:=true                        {yes; quit}
    else
      begin                             {no...}
      Start:=Start+Pos+Plength-1;       {update search starting position}
      Pos:=strpos(str(Source,Start,strlen(Source)-Start+1),Pattern);
                                        {where in THIS PART is Pattern?}
      end;  {else}
    end;  {while}
  if Found<Occurrence then                  {did we exit loop for failure?}
    strpos2:=0                          {yep...}
  else
    strpos2:=Start+Pos-1;               {no, we found the one sought}
  end;  {Occurrence>1}
```

As each occurrence is found, the new value of `Start` specifies the remaining portion of the string to be searched.

String-to-Numeric Conversions

The `strread` function converts a string expression into any type. A `strread` can do anything with a string that a `read` can do with a file with the exception of end-of-line-related operations. When reading an integer or an enumerated-type item, the string must evaluate to a valid value or error −10 will result[2]. Note that enumerated types include the `boolean` type, because `boolean` is just an enumerated type, defined `boolean=(false,true)`, which "comes with the system."

```
error -10: bad input format
```

The `strread` procedure requires at least four parameters. They are:

`strread(`⟨*string to read from*⟩`,`⟨*starting position*⟩`,`⟨*next character to read*⟩`,`
⟨*variable 1*⟩`, ... ,`⟨*variable n*⟩`);`

2 The system reports `escapecode` = −10, but technically, this only means that some kind of I/O error took place, and thus `ioresult` is nonzero. At this point, `ioresult` is examined by the system (it equals 14) in order for the system to print the "bad input format" message.

A description of these parameters follows:

1. The string expression in which certain characters are to be converted into a number (either integer or real values can be read), an enumerated type, etc.
2. The starting index. This integer expression specifies where in the source string read should begin. It must be in the range from 1 to the current length of the source string.
3. The "next" character. Upon completion of the `strread` procedure, this integer variable contains the next character to be read after all the variables (see Step 4, below) have been assigned.

   For example, if you were reading the string `'123 456'` into a single variable, and the starting index was 1, this integer, specifying the "next" character would be 4. This is because after reading the characters `'123'` and converting them to the integer 123, character 4 is the next one to process. So, the next time through the loop, the *second* parameter would be set to 4, reading started there, and "next" would be assigned 8. Observe:

```
program StringRead(output);
var
  Strng:                string[80];
  Start, Next:          integer;
  Number:               integer;
  Color:                (Red, Green, Blue);
  Truth:                boolean;
begin
Strng:='123 red true 45 green true 6789 blue false ';
Start:=1;
while Start<>strlen(Strng) do
  begin
  strread(Strng, Start, Next, Number, Color, Truth);
  writeln(Number, ' ', Color, ' ', Truth);
  Start:=Next;
  end;
end.
```

   This program prints:

```
 123 RED TRUE
  45 GREEN TRUE
6789 BLUE FALSE
```

4. After the first three parameters specified above, there must exist one or more variable names, into which the `strread` procedure places values converted from the string.

A number returned by the `strread` function will be converted to and from scientific notation when necessary. For example:

```
strread('123.4E3',1,NextChar,NumValueRead);
writeln(NumValueRead);
```

Prints: `123400`

The following program converts a fraction into its equivalent decimal value.

```
$switch_strpos$
program ValFrac(input, output);
var
  Fraction:                              string[255];
  Delimiter, Numerator, Denominator:     integer;
  NextChar:                              integer;
begin
write('Enter a fraction (e.g., 3/4): ');
readln(Fraction);
Delimiter:=strpos(Fraction,'/');
Fraction[Delimiter]:=' ';       {remove slash so STRREAD will work}
strread(Fraction,1,NextChar,Numerator, Denominator);
Fraction[Delimiter]:='/';       {put it back so the fraction looks right}
writeln(Fraction,' = ',Numerator/Denominator:30:15);
end.
```

Similar techniques can be used for converting feet and inches to decimal feet, or hours and minutes to decimal hours.

Character-to-Numeric Conversions

The `ord` function converts a single character into its equivalent numeric value; that is, its ASCII value. The number returned is in the range 0 to 255. For example:

```
writeln(ord('A'));
```

Prints: `65`

The next program prints the value of each character in a name.

```
program Ord_(input, output);
var
  Strng:        string[255];
  I:            integer;
begin
write('Text: ');
readln(Strng);
for I:=1 to strlen(Strng) do
  write(ord(Strng[I]):0,' ');
writeln;
end.
```

Entering the name "JOHN" will produce the following.

```
74 79 72 78
```

Numeric-to-String Conversions

The `strwrite` function converts the value of a numeric or enumerated-type expression into a string of characters (again, "enumerated type" includes `boolean`). A string representing a number contains the same numeric characters (digits, decimal point, and/or exponent) that appear when the numeric variable is printed. For example:

```
strwrite(Strng,1,I,1000000,' ',true);
writeln(1000000,' ',true,' ',Strng)
```

Prints: `1000000 TRUE      1000000 TRUE`

A function could be defined which takes a real number as an argument (that way, integers could be passed to it, too) and returns an appropriate-looking string. You probably would `strwrite` the number into the string with a large enough format specifier so as to avoid scientific notation until absolutely necessary. For example:

```
strwrite(Strng,1,Next,RealNumber:31:15);
```

After the number (which can have up to 15 digits on each side of the decimal point before resorting to scientific notation) is in the string, you could then remove the leading spaces, trailing zeroes, and possibly a trailing decimal point from the string, and you're done.

Numeric-to-Character Conversions

The `chr` function converts a number into an ASCII character. The number must be an integer, and the value must be in the range 0 through 255. For example:

```
writeln(chr(97),chr(98),chr(99));
```

Prints: `abc`

The next program converts the numeric values in an array constant to characters.

```
program Chars(input,output);
type
  CharArrayType=array[1..15] of 0..255;
const
  CharArray=    CharArrayType
                  [34,130,89,111,117,32,103,111,116,32,105,116,33,128,34];
var
  I:            integer;
begin
for I:=1 to 15 do
  write(chr(CharArray[I]));
writeln;
end.
```

String Repeat

The `strrpt` function returns a string created by repeating the specified string a given number of times.

```
writeln(strrpt('* *',10));
```

Prints: * ** ** ** ** ** ** ** ** ** *

This function can be used when centering titles. The algorithm is:

1. Subtract the length of the title from the width of the printer/display device to find out how much space is *not* taken up by the title.
2. Divide this amount by two to find the amount of space which should be on the left side of the title.
3. Print that amount of space, followed by the title. The title will be centered.

For example:

```
Title:='⟨any text, as long as it's narrower than the printer⟩';
writeln(strrpt(' ',(PrinterWidth-strlen(Title)) div 2),Title);
```

Note that this will work in the intuitive way for all titles shorter than the printer is wide, *as long as there are no unprintable characters in it* (for example, underlining the title requires a `chr(132)` at the beginning and a `chr(128)` at the end). To take care of this case, just subtract 1 from the length of the title for every character not in the range `' '..'▒'`, or `chr(32)..chr(127)`.

Trimming a String

The `strltrim` and `strrtrim` functions return string with all left (leading) and right (trailing) blanks (ASCII spaces) removed, respectively.

```
writeln('*',strltrim('    1.23    '),'*');
writeln('*',strrtrim('    1.23    '),'*');
writeln('*',strltrim(strrtrim('    1.23    ')),'*');
writeln('*',strrtrim(strltrim('    1.23    ')),'*');
```

Prints:

```
*1.23    *
*    1.23*
*1.23*
*1.23*
```

Combining Strings

There are several ways to combine multiple strings into a single string:

Concatenation	This operator works with any number of string expressions.
`strappend`	This procedure appends one string expression to a string.
`strinsert`	This procedure inserts one string into another at any point.

Concatenation

Note that the concatenation operator is just that–an *operator*–which means that it is placed between the operands it is to combine (infix order). As mentioned, it can combine any number of string expressions:

```
Concatenation:='String 1'+'String 2';
All:=A+'another'+(strrpt(' ',Width div 2)+str(Strng,2,5))+'and '#7+Strng;
```

Appending Strings

This procedure requires a string variable as the first parameter; the second parameter may be any kind of a string expression. Upon completion, the string variable has the value it had before with the string expression concatenated to it at the right end.

```
String:='Pascal ';
strappend(Strng,'strings');
writeln(Strng);
```

Prints: `Pascal strings.`

Inserting in the Middle

This procedure requires a string variable, a string expression, and an index into the string variable. The procedure causes the string expression to be inserted into the string variable at the specified (index) point.

```
Strng:='Thus';
strinsert(Strng,'esau',3);
strinsert(Strng,'r',7);
writeln(Strng);
```

Prints: `Thesaurus.`

Replacing/Appending and Conversion Between Strings and PACs

There is another string-related procedure called `strmove` which allows several operations to take place:

- You can append characters to the end of a string (e.g., "`bring`"→"`bringing`");
- You can replace characters in a string one-for-one with other characters (e.g., "`inside`"→"`in the`");
- Both of the above–replacing characters and extending the string–simultaneously (e.g., "`sheaf`"→"`sheaves`");
- Convert a PAC variable[3] to a string, and vice versa, without having to move the characters one at a time.

3 A "PAC" variable is a `packed array [1..n] of char`. Note that the array must be packed, and the first subscript of the array must be 1.

The procedure `strmove` takes five parameters. First, the number of characters to move from the source to the destination. Next, for both the source and the destination, the entity and the index into the entity. For example:

```
strmove(Nchars,SourceExpr,SourcePos,DestVar,DestPos);
strmove(4,A,5,B,6);          { Move 4 characters, starting with A[5], into }
                             { B, starting at position 6.                  }

S:='Pal';
strmove(1,'that',4,S,2); {Move 1 character, starting with the 4th character}
                         {of 'that', into S ('Pal'), starting at Position 2}
```

Reducing Strings

You can delete characters in a string in any one of several ways:

- Deleting one or more characters from the "middle" (the "middle" could extend to either end, or conceivably to both ends, deleting the whole thing),
- Deleting one or both ends of the string simultaneously (the deleted portions could conceivably touch, deleting the whole thing),
- Trimming leading or trailing blanks (we saw this before).

Deleting Characters from the Middle

The `strdelete` procedure deletes a specified number of characters from the middle of a string. You specify the string to be reduced, where to start deleting characters, and how many characters to delete:

```
Strng:='strings';                    {Strng now equals 'strings',}
strdelete(Strng,7,1);                {Strng now equals 'string',}
strdelete(Strng,2,2);                {Strng now equals 'sing',}
strdelete(Strng,strlen(Strng),1);    {Strng now equals 'sin',}
strdelete(Strng,1,13 mod 4);         {Strng now equals 'in',}
strdelete(Strng,2,2);                {Strng now equals '',}
```

Deleting Both Ends

In order to delete zero or more characters from both ends of a string simultaneously, you have to get a trifle cagey. You don't really *delete the ends*, you *retain the middle*; that is, you assign the variable the value of a substring from the middle:

```
Strng:='antidisestablishmentarianism';
Strng:=str(Strng,8,9);                    {Strng now equals 'establish',}
```

Trimming Blanks

We discussed these functions before (see "Trimming a String", above). The functions `strltrim` and `strrtrim` remove leading and trailing blanks, respectively.

User-Defined String Functions

Although there are several string functions available in Series 200 Pascal, there are several more which are not supplied with the language which can be very useful.

Note

When creating special string functions, testing should include passing the null string ('') to the function. The null string is a valid string and may get passed to the function.

Case Conversion

Often, you may want to convert the letters in a string–keyboard input, for example–to uppercase or lowercase letters. This is quite an easy thing to do, since the ASCII values (the "ord") of uppercase letters differ from the ASCII values of the corresponding lowercase letters by 32. Below is the algorithm for converting to uppercase:

```
for I:=1 to strlen(Strng) do
  begin
  Character:=Strng[I];                    {avoid subscripting multiple times}
  if (Character>='a') and (Character<='z') then
    Strng[I]:=chr(ord(Character)-32);
  end;  {for I}
```

The algorithm for converting to lowercase is very similar; you just *add* 32, rather than subtracting 32:

```
for I:=1 to strlen(Strng) do
  begin
  Character:=Strng[I];                    {avoid subscripting multiple times}
  if (Character>='A') and (Character<='Z') then
    Strng[I]:=chr(ord(Character)+32);
  end;  {for I}
```

Note: both of these algorithms can be sped up having the compiler option `$Partial_eval$` in effect.

String Reverse

A string reversal function returns a string created by reversing the sequence of characters in the given string. For example, reversing `'abc'` results in `'cba'`. Again, the algorithm is elementary:

```
Length:=strlen(Strng);
LengthPlus1:=Length+1;        {avoid adding 1 every iteration...}
for I:=1 to Length div 2 do
  begin
  Temp:=Strng[I];
  RightChar:=LengthPlus1-I;
  Strng[I]:=Strng[RightChar];
  Strng[RightChar]:=Temp;
  end;
```

Note that when the string has an even number of characters in it, all appropriate pairs of characters are switched in position, but when the string has an odd number of characters in it, the middle character is never addressed. This is fine; it doesn't *need* to be addressed, because the middle character is the middle character, regardless of which end you start from.

If you incorporated the above algorithm into a function called `strrev`, the following statement:

```
writeln(strrev('Straw? No, too stupid a fad. I put soot on warts.'));
```

would print:

```
.straw no toos tup I .daf a diputs oot .oN ?wartS
```

A common (but inefficient) use for the string reversal function is to find the last occurrence of an item in a string. Assume again that a function `strrev` is defined which returns the reversed argument.

```
$switch_strpos$
      .
      .
      .
var
  Strng, LastItem:  string[80];
  Delimiter:        string[1];    {must be a string; STRPOS doesn't like CHAR}
  LastDelim:        integer;
      .
      .
      .
Strng:='Now is the time for all good men to come to the aid of their country.';
Delimiter:=' ';
LastDelim:=strlen(Strng)-strpos(strrev(Strng),Delimiter)+1;
LastItem:=str(Strng,LastDelim+1,strlen(Strng)-(LastDelim+1)+1);
writeln('The last item is "',LastItem,'".');
```

Displays: `The last item is "country."`

Search-and-Replace Operations

A commonly used operation when dealing with strings is this: "I want to replace each one of *these* in this string with one of *those*." This very useful function entails several sub-operations:

1. Find, in the main string, the first occurrence of the "old" string (that string which is to be replaced, hereafter called ⟨*old*⟩).
2. Delete that occurrence of ⟨*old*⟩, and insert one occurrence of the string which is to replace it (hereafter called ⟨*new*⟩). Note that this must be a deletion followed by an insertion; it cannot be a "direct replacement", because ⟨*new*⟩ may be a different length than ⟨*old*⟩.
3. Starting from the first character *after the end* of the newly-inserted ⟨*new*⟩, search for another occurrence of ⟨*old*⟩. You cannot just start searching again from the beginning of the main string, because it is perfectly legal for ⟨*new*⟩ to contain one or more occurrences of ⟨*old*⟩. If this was the case, searching from the beginning of the main string would result in either (1) an infinite loop, if ⟨*new*⟩ = ⟨*old*⟩, or (2) a string overflow error if `strlen(`⟨*new*⟩`)>strlen(`⟨*old*⟩`)`.
4. Repeat steps 2 through 3 until there are no more occurrences of ⟨*old*⟩ in the searchable section of the main string.

Taking these things into account, the following code segment accomplishes the desired task. Assume that the type `Str255` has been defined as `string[255]`, and that `$switch_strpos$` is in effect. `Strng` is the main string in which the replacements take place; `Old` and `New` have their intuitive meanings.

```
$switch_strpos$
      .
      .
      .
var
  LengthOfStrng:        integer;
  LengthOfOld:          integer;
  LengthOfNew:          integer;
  Pos, Temp:            integer;
      .
      .
      .
if (Strng='') or (Old='') then {do nothing}
else begin
  LengthOfStrng:=strlen(Strng);      { \                                  }
  LengthOfOld:=strlen(Old);          {  > Things go faster this way... }
  LengthOfNew:=strlen(New);          { /                                  }
  Pos:=strpos(Strng,Old);
  while Pos>0 do begin
    Strng:=str(Strng,1,Pos-1)+New+
     str(Strng,Pos+LengthOfOld,LengthOfStrng-(Pos+LengthOfOld)+1);
    LengthOfStrng:=LengthOfStrng-LengthOfOld+LengthOfNew;
    Temp:=Pos+LengthOfNew;
    Pos:=strpos(str(Strng,Temp,strlen(Strng)-Temp+1),Old);
    if Pos>0 then Pos:=Pos+Temp-1;
  end;  {while}
end;  {else begin}
```

Sections of Strings

This section just discusses how to get the more common parts of strings in the easiest way.

Left Part

To get the left part of a string, up to and including character *n*, do the following:

```
Strng:=str(Strng,1,n);
```

Right Part

To get the right part of a string, from character *n* to the end, do the following:

```
Strng:=str(Strng,n,strlen(Strng)-n+1);
```

Middle Part: Point A through Point B

To get the middle part of a string when you know the starting and ending positions (*start* and *finish*, respectively), but do not know how many characters this includes, do the following:

```
Strng:=str(Strng,start,finish-start+1);
```

Programming With Files

Chapter 15

Introduction

The File System of your Pascal system organizes and accesses information which is stored in files on mass storage devices. This section describes how the information is organized and accessed. It consists of the following major discussions:

- Overview of mass storage, including descriptions of files and volumes.
- Techniques for using item-oriented files.
- Techniques for using line-oriented files.
- More details of Pascal Workstation files.

If you are already familiar with files and volumes, you may want to just scan the drawings in the overview section. The "techniques" sections give several examples and some good advice about using Workstation files. The "More Details" section is a more in-depth look at the facts about the Workstation File System.

Overview of Files

This section describes several concepts relating to the use of mass storage files.

Primary versus Secondary Storage

Your computer has built into it a substantial amount of very high-speed memory called random-access memory, or RAM. This memory is called *primary storage* to distinguish it from mass storage, which is called *secondary storage*. Normally, data processed by the computer must be first placed in primary storage. (The term "data" is used here broadly to signify any information processed by the computer; thus, programs are data, too.)

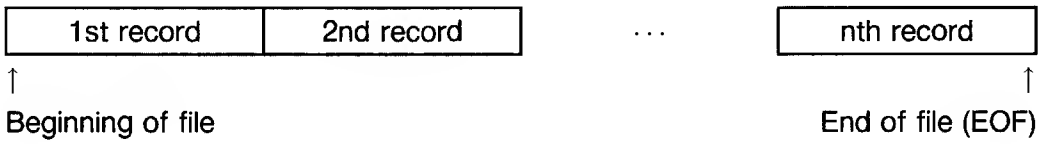
Information not immediately needed by the computer is kept in secondary storage. Mass storage devices are typically less expensive to maintain, are non-volatile (information is not lost when power is removed), and have much greater capacities (hence the term "mass").

What Is a File?

Good question. A file is a logically defined storage area set aside for the temporary or permanent storage of a collection of similar data items. Files consist of two main parts:

- A description (a name, type, size, etc. in a mass storage *directory*).
- Some data (the actual information it contains).

Here is a conceptual drawing of the data part of a file, showing its sequential nature.

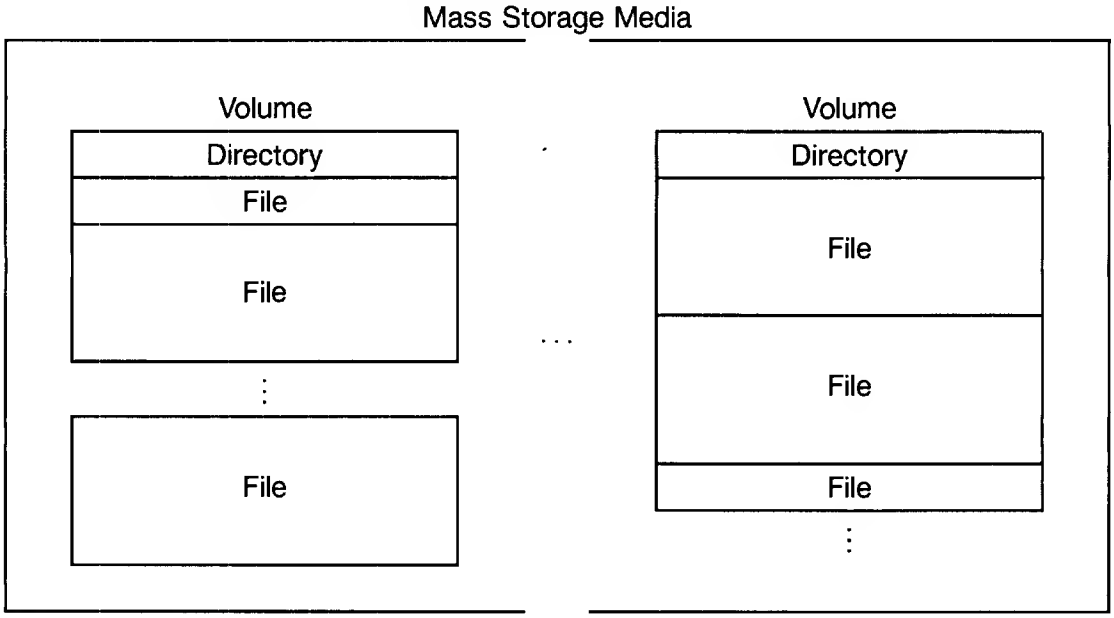


Structure of the Data Portion of a File

In a nutshell, mass storage access consists of the Pascal file system creating a *file* on a particular *volume*, and then writing to or reading from individual records in that file. Before showing examples of that, however, let's complete an overall picture of mass storage by looking at directories and volumes.

Mass Storage Organization (Non-Hierarchical Directories)

Mass storage is organized into volumes, each of which may contain several files. Here is a pictorial representation of the relationship between volumes and files.

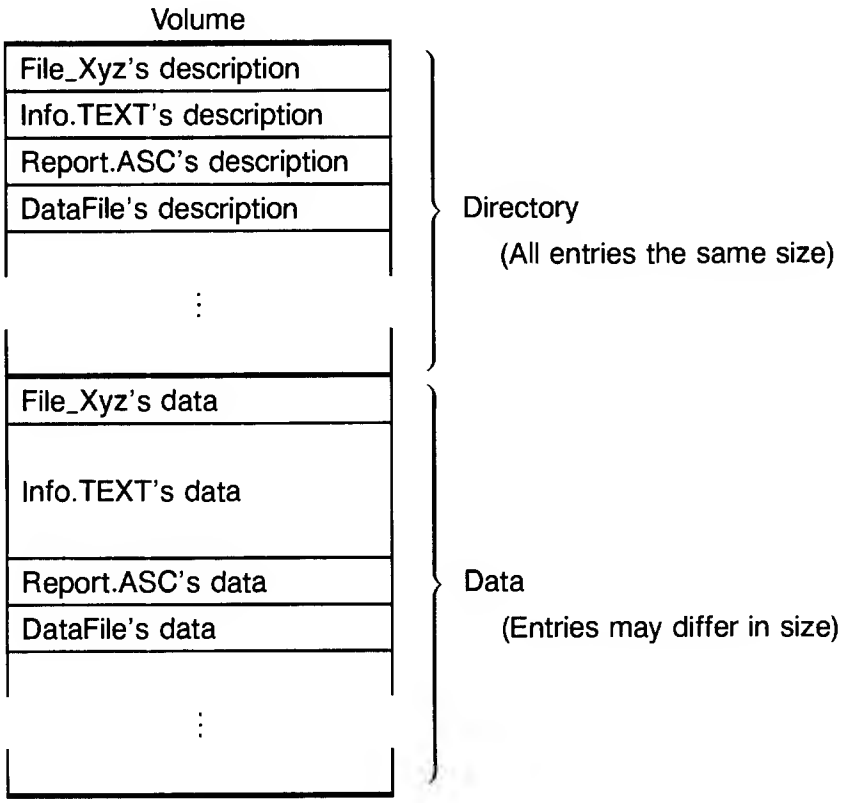


Mass Storage Device Organization

(Hard Discs, Flexible Discs, Cartridge Tapes, etc.)

Volume Structure

The term "volume" was chosen by analogy to a book. A book contains a table of contents and information. Similarly, volumes contain a directory of the files in them, as well as the information in each file. Here is a graphic representation of how volumes are organized.



Directory entries usually contain such information as:

- File name
- File type
- Start location of file (offset from beginning of volume)
- Number of blocks allocated to file
- Current length of file (in bytes)
- Date created
- Date last modified

However, the information contained in directory entries varies with the type of directory (e.g., LIF, WS1.0, SRM).

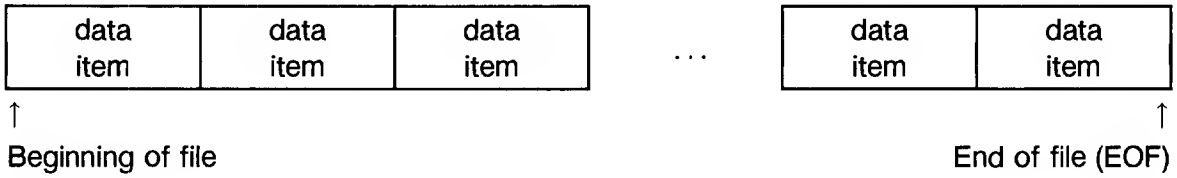
Classifications of Files

There are two main classes of files that the Pascal system can deal with:

- Item-oriented, or "fixed-record" files.
- Line-oriented, or "text" files.

Item-Oriented Files

As the term "item-oriented" suggests, this type of file consists of data "items", each of which has a fixed size. Because of this, the name "fixed-record files" is often used. Records in these files can be any Pascal `type`, such as pre-defined simple types (e.g., `char`, `integer`) or structured types (e.g., `record`, `array`), and only data of that type can be put on the file.

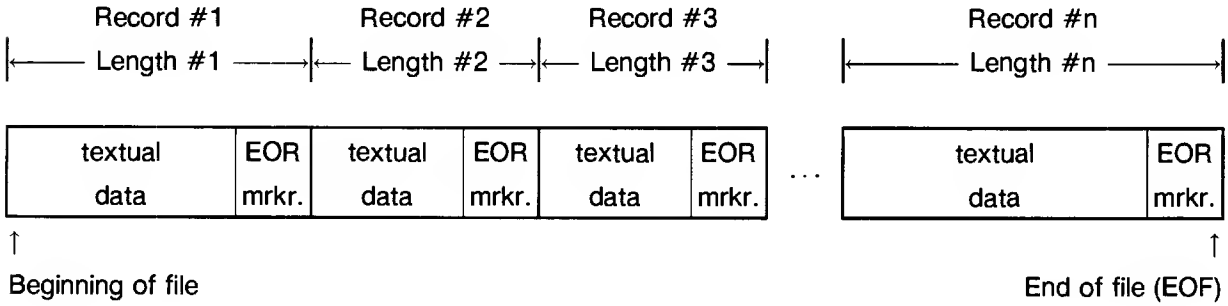


Structure of Item-Oriented ("Fixed-Record") Files

When data is placed onto an item-oriented file, it is written in internal format; bit for bit, the same structure as memory.

Line-Oriented Files

As the term "line-oriented" implies, this type of file consists of records which are lines of text. Another common name for this type of file is "text files." Lines in such a file may vary in length (Length 1, Length 2, etc., in drawing below), so they are terminated by a unique "end-of-record" (EOR) marker. (Note that `.ASC` files have no explicit EOR mark; see "Other Types of Text Files" later in this chapter).



Structure of Line-Oriented "Text" Files

You can write or read either an entire record or part of a record at a time. And when reading, you can determine if you are at the end of a line or at the end of the file.

When data is placed onto an line-oriented file, it is *not* written in internal format; it is formatted to be readable by humans.

In a nutshell, the Pascal system locates a *file* on a particular *volume*, and then can write to or read from that *file*, one *record* at a time.

Item-Oriented Files

In the previous section, a very brief mention was made of a "`file of` ⟨*type*⟩." Here we will delve further into what they are and why they are useful.

In past examples, the text files were opened for writing with `rewrite`, and opened for reading with `reset`. Since text files are line-oriented, they were written to with `writeln` and read from with `readln` (although `write` and `read` could have been used in addition).

With item-oriented files, `writeln`, and `readln` won't work. To open a file, you use an `open`, `append`, `rewrite` or `reset` statement. To write to and read from a file, you use `write` and `read`. Here is one advantage to item-oriented files: they can be *random-access* files (another term is *direct-access*). That is, to read or write item 54, you don't need to read or write items 1 through 53.

Creating and Writing to an Item-Oriented File

It is as simple to create a file of some type of data-type entries as it is to create text files. One slight difference, though, between `text` files and `file of` ⟨*type*⟩ files is that `file of` ⟨*type*⟩ files must have ⟨*type*⟩ defined by the user if it is not a standard Pascal `type`. The declaration `text` is a standard Pascal type, and so the user doesn't need to worry about defining it.

In a `file of` ⟨*type*⟩, ⟨*type*⟩ can be any valid Pascal `type` constructor, except those containing files; files cannot be nested. For example, you can use predefined data types to define the items for a file:

```
var
  FirstFile:          file of char;
  SecondFile:         file of integer;
  ThirdFile:          file of real;
```

You can also define your own data items with which to define a file:

```
type
  GI=                 record
                        Name:          string[30];
                        Rank:          (Private, Lieutenant, Captain);
                        SerialNumber: string[13];
                      end;
  Squad=              array [1..11] of GI;
var
  GIfile:             file of GI;
  SquadFile:          file of Squad;
```

Write the following small program and execute it[1]:

```
program One;
var
  Test:          file of integer;
  I:             integer;
begin
rewrite(Test,'#3:Test');
for I:=1 to 10 do
  write(Test,I*I);
close(Test,'save');
end.
```

1 This and the following examples assume that you have a mass storage device at unit #3. If this is not the case for your system, modify the example appropriately for your hardware. (The Filer's Volumes command shows you the units that are on-line.)

The variable `Test` is a file all of whose entries are of type `integer`. This means that whenever the program uses the variable `Test`, it is referring to a file which is accessed as integers being written to and read from it.

In the actual execution of the program, the `rewrite` statement causes a file called "`Test`" to be created on unit #3 (if it does not already exist). Since the file variable used in that statement is `Test`, and since the file variable `Test` is declared in the program to be of type `file of integer`, the computer assumes that the file called `Test` is of type `file of integer`. Integers are then written, one at a time, to the file.

Now enter the Filer and type

L :Return

You should get something like the following:

```
V3:                Directory type= LIF level 1
created 23-Jan-85 10:43:41 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test                       1         40 14-Mar-85
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=1 unused=1043 largest space=1043
```

The file size is 40 bytes, as indicated by the Filer. This is as it should be, since integers are four bytes (these are 32-bit integers) apiece, and we wrote 10 integers to the file. Note that there is no need for anything like the EOR markers (length headers or carriage returns) in this type of file, as there is in text files. The reason for this is that all the items in the file are the same size, so a simple multiply of the item number sought minus one (because the first item is record 1) by the size of the item gives the location of an item in the file. In text files, the lines can vary in length, so there needs to be some kind of delimiter separating them.

Reading Sequentially From a File

Now that we've written something to the file, let's read it back. Enter this next program into your computer, then compile and run it.

```
program Two(output);
var
  Test:          file of integer;
  RecNumber:     integer;
  Value:         integer;
begin
reset(Test,'#3:Test');
for RecNumber:=1 to 10 do
  begin
  read(Test,Value);
  writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
  end;
close(Test);
end.
```

When the program runs, it reads and prints sequential records on the file.

Detecting the End of the File

The end-of-file condition may be a bit anti-intuitive when dealing with this type of files, where items can be read in any order, because you're may not necessarily be done with the file just because you read the last record. But you can read a random-access file sequentially, in which case the `eof` function works the normal way. Suppose you want to read the file like this:

```
program Three(output);
var
  Test:          file of integer;
  RecNumber:     integer;
  Value:         integer;
begin
reset(Test,'#3:Test');
RecNumber:=0;
  repeat
  read(Test,Value);
  RecNumber:=RecNumber+1;
  writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
  until eof(Test);
close(Test);
end.
```

This program will read the file from start to finish. Record #10 is the last record, so when that is read, `eof(Test)` goes `true` and the program finishes. (Note that a `while not eof` loop is a bit better than a `repeat until eof` loop, because the former can also handle an empty file.)

Now consider the following program. It reads the file, but *in a user-specified order.*

```
program Four(input, output);
var
  Test:          file of integer;
  RecNumber:     integer;
  Value:         integer;
begin
open(Test,'#3:Test');
  repeat
  write('Record number to read: ');
  readln(RecNumber);
  seek(Test,RecNumber);
  read(Test,Value);
  writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
  until eof(Test);
close(Test);
end.
```

The `eof` function still goes true when record #10 is read, even if it is the first one you read. This is probably not what you want. Thus, you will probably want to determine the exit-the-loop condition on something other than actual end-of-file, since the "end of processing" may occur anywhere.

Also note that the above program crashes if you specify a record number outside the range of 1 through 10. This is certainly not user-friendly. You can put in a check for this by using the function `maxpos`, which tells you what is the highest-numbered record in the data file (this does not work for text files).

```
program Five(input, output);
var
  Test:          file of integer;
  RecNumber:     integer;
  Value:         integer;
begin
open(Test,'#3:Test');
  repeat
  write('Record number to read: ');
  readln(RecNumber);
  if (RecNumber>=1) and (RecNumber<=maxpos(Test)) then
    begin
    seek(Test,RecNumber);
    read(Test,Value);
    writeln('Record number ',RecNumber:0,' contains ',Value:0,'.');
    end
  else
    writeln(#7'Illegal record number specified.');
  until eof(Test);
close(Test);
end.
```

After being satisfied that you understand the above examples, remove any leftover files that remain from their execution from the disc now, so they won't clutter the Filer listings for future examples.

In the next section, line-oriented files, or *text* files, are discussed.

Line-Oriented (Text) Files

This section deals with files whose organization is oriented toward lines of text. These lines of text have different maximum lengths, depending on the file type:

.TEXT	1023 characters.
.ASC	32 767 characters.
Data	No limit.

See the section "Other Types of Text Files," later in the chapter for more information.

Since the items in line-oriented files need not be the same length, some mechanism must exist whereby each line's length is determined at the time of reading or writing. This concept is different from *item-oriented* files, wherein all the items are the same structure *and* size. That type of file was discussed earlier in this chapter.

Creating a File

It is a simple thing to create a file on a mass storage device. Write the following small program and execute it:

```
program Six;
var
  Test:          text;
begin
rewrite(Test,'#3:Test2.TEXT');
close(Test,'save');
end.
```

The variable `Test`, as you can see from the declaration, is of type `text`. This means that whenever the program uses the variable `Test`, it is referring to a file which can be accessed as "lines" of text. The `rewrite` statement associates the variable `Test` with the file named `Test2.TEXT`. (Note that if `Test2.TEXT` does not already exist, the file system creates it automatically.)

In the actual execution of the program, the `rewrite` statement creates a file called "`Test2.TEXT`" on unit #3. Since the file variable used in that statement is `Test`, and since the file variable `Test` is of type `text` (as per the declaration part of the program), the computer knows that the file called `Test2.TEXT` can contain `text`.

Files are sometimes used as temporary entities, existing only for the duration of the program. This is what the Pascal system assumes, unless you tell it otherwise. For this reason, the `close` statement is included in the program. The `close` tells the computer to "close" the file; that is, disassociate the actual file from the program currently executing, and the program disavows any knowledge of the file's actions or attributes. The file to be closed is specified by the first parameter. What is done with the file at its closure is specified by the second parameter; `'save'` in this case (note that `'save'` is a string parameter), which makes the file "permanent" on the mass storage device.

If the `close` statement doesn't have a second parameter, the file is closed with the same attributes as it had before the opening of the file with the `rewrite`. This file didn't exist before the `rewrite`; thus, if you closed the file without the second parameter, it would be gone at the end of the program. Again, the `'save'` string[2] as the second parameter causes the file to remain in existence after the program ends.

Now enter the Filer and type

L :Return

which tells the computer to list ("L") the directory of the default (":") volume. You should get something like the following:

```
V3:                Directory type= LIF level 1
created 23-Jan-85 11:11, 9 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test2.TEXT                0          0 12-Mar-85
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=0 unused=1044 largest space=1044
```

You may have more entries listed on your screen, but let's only deal with the `Test2.TEXT` file. Note that its name, size (in both blocks and bytes), and date of last change are listed. This confirms it: you *did* create a file with your program.

Writing to a File

Go back into the Editor and modify your program. Place these two statements immediately after the `reset` statement:

```
writeln(Test,'Printing one line...');
writeln(Test,'...and another.');
```

This tells the program to write, to the file indicated by file variable `Test`, the two strings "`Printing one line...`" and "`...and another.`". Note that the single quote marks are merely delimiters of the strings; they just specify where the strings start and stop. They are not considered part of the strings, and thus are not written to the file.

Now compile and execute the new version of the program. After it finishes, enter the Filer again and look at the volume listing (do the L :Return command again).

```
V3:                Directory type= LIF level 1
created 23-Jan-85 11:11, 9 block size=256
 Storage order
...file name....    # blks    # bytes  last chng

Test2.TEXT                8       2048 12-Mar-85
FILES shown=1 allocated=1 unallocated=79
BLOCKS (256 bytes) used=8 unused=1036 largest space=1036
```

2 The second parameter on the `close` statement may be uppercase or lowercase; letter case is completely ignored. The string `'lock'` does the same action as `'save'`.

The file is a different size this time. This is not surprising, considering we put nothing in the file the first time, and thirty-five characters in it the second time. But why did the file size increase so dramatically? We put thirty-five characters into the file, and the file size increases by more than two *thousand* bytes?

The answer lies in the definition of .TEXT files. .TEXT files have these two characteristics:

- A file whose name at creation time ends in .TEXT contains a "header"–a 1K-byte area which does not contain text, but is an area for carrying information about the file. For example, when you create a .TEXT file with the Editor, margin information, markers, and all the other environment information are stored in this area.
- A .TEXT file takes up space only in increments of 1K bytes. Therefore, when we wrote the thirty-five bytes into the file, it took 1024 bytes. If we added more and more bytes to this file, its size would not increase until more than 1024 bytes (excluding the 1K-byte header) were written, at which time another 1024 bytes would be appropriated.

Reading a Text File with the Editor

Did the characters actually make it to the file the way we wanted them to?

To see if the two lines of text actually made it onto the file, let's try to read the file into the Editor and see if the text is there. (Make sure you store program Six first so you don't lose it.) When you enter the Editor, specify Test.TEXT as the file name to edit. Sure enough, the Editor displays:

```
Printing one line...
...and another.
```

This file can be edited in the usual way, re-stored, etc., just like a file which was created by the Editor. In fact, you cannot tell by looking at any particular .TEXT-type file with the Editor whether it was created programmatically, as above, or with the Editor.

Reading a Text File with a Program

Now that you've written something in the file, how do you read it back later? This is also an easy task. Enter this next program into your computer, then compile and run it.

```
program Seven(output);
var
  Test:          text;
  Line:          string[80];
begin
reset(Test,'#3:Test2.TEXT');
readln(Test,Line);
writeln(Line);
close(Test);
end.
```

When the program runs, it prints:

```
Printing one line...
```

It only read and printed the first line of the file; that's all we told it to do. To read and print the second line, merely add the following two lines right before the `close` statement:

```
readln(Test,Line);
writeln(Line);
```

With this modification, the program will read the first line from the file, print it, read the second line from the file, and print it. But now add a third pair of statements:

```
readln(Test,Line);
writeln(Line);
```

Now the program will attempt to read and print three lines from the file. But the file only contains two lines! What will happen when the program attempts to read the "third" one? Let's run the program and find out.

The computer displays the following:

```
Restart with debugger ?
Printing one line...
...and another.

-----------------------------------------------
error -10: tried to read or write past eof
PC value:    -1390528
```

What happened was this:

1. The computer successfully reads the first line of text from the file and prints it out.
2. The computer successfully reads the second line of text from the file and prints it out.
3. The computer tries to read the (nonexistent) third line of text from the file. However, there is no third line of text, so the statement is impossible to carry out. The computer considers this an error, and informs you by doing the following things:
   a. If your computer has a beeper, it beeps at you.
   b. A row of minus signs is printed to draw your attention to the error message to follow.
   c. The error message is printed. The "`error -10`" part of the message indicates that the problem was an I/O error. However, the message printed is not "`I/O error`"; the next part of the message tells you what kind of I/O error occurred: you `tried to read or write past eof`. The "`eof`" means "end of file".
   d. The PC (program counter) value is printed. This can be used for debugging, but we will not address that here.
   e. You get the option of restarting the program from the Debugger. The Debugger allows you to execute the program piece by piece, look at variables' values, etc. See the *Debugger* chapter of the *Pascal Workstation System* manual for more information on this.

Fine. Now we know what happened, but how can we stop it from happening again? And if we are trying to read a file someone else created, or even one that we created, we may not know how many records we'll have to read. There could be *any* number of records in a file (within the size limit of the physical volume).

Detecting the End of the File

The boolean function called `eof`, which tells you when the end of the file has been reached, also works with line-oriented files. Modify your second program so that it looks like the following:

```
program Seven(output);
var
  Test:          text;
  Line:          string[80];
begin
reset(Test,'#3:Test2,TEXT');
while not eof(Test) do        {new}
  begin                       {new}
  readln(Test,Line);
  writeln(Line);
  end;                        {new}
close(Test);
end.
```

The `while` statement checks every time through the loop (*before* executing the loop) whether or not the end of the file has been reached. The loop is only executed if the end of the file has *not* been reached. The following steps are executed:

1. Open the file.
2. The file is not a completely empty file, so `eof(Test)` is initially `false`.
3. Test for EOF. Read the first line ("`Printing one line...`") and print it. The end of the file has not been reached yet
4. The loop iterates, since the end of the file was not found in step 3.
5. Test for EOF. Read the second line ("`...and another.`") The end of the file is found. Print the line.
6. The loop does not iterate again because the end of the file was found at the end of the second line of text.
7. Close the file.

Detecting the End of a Line

In addition to using the `eof` function, you can also use the function `eoln` in conjunction with text files. Obviously, and end-of-*line* function makes no sense in files which are not *line*-oriented.

If you read text from a Text file into a string or a packed array of characters, the variable will be filled until either:

- An EOLN is reached
- The variable is filled to capacity

To read a line from a text file which is longer than the string you are putting it in, you can read the line in pieces, using `read`. When you get to the end of a line, use `readln` to go to the next line. Outside this loop, you use an EOF loop like before. Thus, you need an EOLN loop inside an EOF loop:

```
program Eight(output);
var
  Test:          text;
  Line:          string[4];
  I:             integer;
begin
reset(Test,'#3:Test2.TEXT');
while not eof(Test) do
  begin
  while not eoln(Test) do
    begin
    read(Test,Line);
    write(Line);
    for I:=1 to 100000 do;      {make a noticeable wait}
    end;
  readln(Test);
  writeln;
  end;
close(Test);
end.
```

When you run this program, notice that four-character (or less) pieces of the lines are read and printed. When the end of the line is reached, it is caught by the `eoln` function.

Other Types of Text Files

In case there is a contradiction/ambiguity starting to loom in your mind, let us define more precisely the two definitions of the phrase "text file." Here are the two definitions:

1. The Pascal language's definition of "text file" is any file which is declared as type `text`, as opposed to `file of` ⟨*type*⟩, in the declaration section of a routine. This type of file can be written to with `writeln` statements, and read with `readln` statements, whereas `file of` ⟨*type*⟩ files cannot. Also, these files can deal with end-of-line conditions, and data is formatted before writing to the file.
2. Pascal's files which are declared `text` can be created as any of three types (the rest of this section of the chapter elaborates on these various file types):

- If a text file whose name ends with `.TEXT` is created, it will be what is called a TEXT file.
- If a text file whose name ends with `.ASC` is created, it will be an ASCII file.
- If a text file whose name ends with anything else is created, it will be a `Data` file.

Note that the file types are only determined by the file name *at the time of creation*. You can change the name of an existent file to anything you want, so conceivably, you could have an ASCII file called `Fred.TEXT`, or a Text-type file called `Data`, or a Data-type file called `TEXT.ASC`.

In this section, we'll be exploring the different types of physical files, all of which are declared of type `text` in a Pascal program.

Creating ASCII and Data Files

After storing program Seven, get program Six again. Modify the file-opening line such that it looks like this:

```
rewrite(Test,'#3:Test2.ASC');
```

Run the program again, and then modify the line again so that it looks like this:

```
rewrite(Test,'#3:Test2');
```

Run the program again, then enter the Filer and list the directory of unit #3: L :Return.

The Filer listing now looks like this (there may be other entries also, depending on what you've put on your disc):

```
V3:                Directory type= LIF level 1
created 23-Jan-85 10.43.41 block size=256
 Storage order
...file name....     # blks     # bytes  last chng

Test2.TEXT                8        2048 13-Mar-85
Test2.ASC                 1         256 13-Mar-85
Test2                     1          37 13-Mar-85
FILES shown=3 allocated=3 unallocated=77
BLOCKS (256 bytes) used=10 unused=1034 largest space=1034
```

Note that although the same text was written to all three files, every one's file size is different. The reason is that the other file types–ASCII, indicated by a .ASC suffix at creation, and Data, indicated by no suffix at creation–have their special characteristics, just as the .TEXT files, mentioned before.

TEXT-type Files

Files of type .TEXT have the following characteristics:

- A file whose name at creation time ends in .TEXT contains the "header" area for carrying information about the file.
- Line-endings are noted by carriage-returns.
- A .TEXT file takes up space only in increments of 1K bytes, and any unused space is zeroed.
- Logical end-of-file is specified by the first character in a line being chr(0).

Note that you cannot reliably put characters whose ord is less than 32 into a .TEXT file, because some of these characters are used in the file and have special meanings.

In our .TEXT file, the actual bytes placed in the file, excluding the header, are:

1. The characters "Printing one line...".
2. A carriage-return (chr(13)), indicating the end of a line of text.
3. The characters "...and another.".
4. Another carriage-return, indicating the end of another line of text.
5. ASCII nulls (chr(0)) for the remainder of the 1K-byte block.

ASCII Files

Files of type .ASC have the following characteristics:

- Lines are specified by two-byte length headers specifying actual length, and lines (in the file only) are padded to an even length.
- A .ASC file takes up space only in increments of 256 bytes.
- End-of-file is specified by two consecutive chr(255)s, which is equivalent to a 16-bit, twos-complement value of −1.

In our ASCII file, the actual bytes placed in the file are:

1. A two-byte (16-bit) length header, indicating the length of the upcoming line. Since our first line has 20 characters, the length header is 00000000 00010100, or an ASCII null, followed by an ASCII "DC4" character (chr(20)).
2. The characters "Printing one line...".
3. Another two-byte length header. Since our next line has 15 characters, the length header is 00000000 00001111, or an ASCII null (chr(0)), followed by an ASCII "shift-in" character (chr(15)).
4. The characters "...and another.".
5. A pad character (ASCII blank; chr(32)) to cause the next length header (if any) to start on an even byte.
6. Another "length header." However, since there are no more lines–the end of the file has been reached–there is a special flag value of this length header. Its value of −1 (two consecutive chr(255)s) tells the computer not to interpret the two bytes as a length header, but as an end-of-file marker.
7. ASCII nulls (chr(0)) for the remainder of the 256-byte block.

Data-type Files

Files of type Data have the following characteristics:

- Line-endings are specified by carriage-returns (chr(13)).
- A Data file takes up only the amount of space it needs, rounded up to the nearest block. That is, it is allocated in blocks, and its physical size remains an integer number of blocks. At file closure, however, the *logical* size is cut back to logical end-of-file, which can occur at any byte in the file.
- End-of-file is specified in the directory.

In our Data file, the actual bytes placed in the file are:

1. The characters "Printing one line...".
2. A carriage-return (chr(13)), indicating the end of a line of text.
3. The characters "...and another.".
4. Another carriage-return, indicating the end of another line of text.

There are intrinsic differences in these file types, and the Pascal operating system keeps track of them in ways other than just their names. As mentioned previously, you can change the name of an existing file to something that can be quite misleading.

To see some of the other ways that Pascal differentiates between file types, go into the Filer and press E :Return. This makes an extended listing. Our volume will look like this:

```
V3:                Directory type= LIF level 1
created 23-Jan-85 10:43:41 block size=256
 Storage order
...file name....     # blks     # bytes    start blk ....last change... extension1
                  type  t-code ..directory info... ....create date... extension2

Test:TEXT                 8        2048          12 13-Mar-85 15:49:38          0
                  Text    -5570                                                  1
Test:ASC                  1         256          20 13-Mar-85 16:21:28          0
                  Ascii       1                                                  1
Test                      1          37          21 13-Mar-85 16:21:41         37
                  Data    -5622                                                  1
< UNUSED >              1034                      22
FILES shown=3 allocated=3 unallocated=77
BLOCKS (256 bytes) used=10 unused=1034 largest space=1034
```

As you can see from the second column, the file type is noted elsewhere than just the name of the file.

Also note that the logical file size of the `Data` file is indicated in the `extension 1` field.

In the next section, the rest of the file-manipulation routines are discussed.

More Details on Programming With Files

This section describes the operation of the Pascal file operations. It discusses the creation and disposition of files, and the basic operations on file data.

Pascal Primitive File Operations

- The following operations put the file into WRITE Mode:

```
REWRITE
OPEN
APPEND
SEEK
PUT
WRITE
WRITEDIR
WRITELN          {see the section on TEXT files}
F^               {if the file is already in WRITE Mode}
```

- The following operations put the file into READ Mode:

```
RESET
GET
READ
READDIR
READLN           {see the section on TEXT files}
```

- The following operations put the file in LOOKAHEAD Mode:

```
F^               {unless the file was in WRITE Mode}
EOF              {unless the file is open for random access}
EOLN             {see the section on TEXT files}
READ             {of multi-character objects from TEXT files, such
                  as strings, PACs, integers, reals, enumerated types,
                  and booleans.}
```

Creating New Files

A file is initially created by the `rewrite`, `open`, or `append` operations. However, `open` and `append` are usually applied to existing files.

These standard procedures each may take one, two or three parameters:

`rewrite(`⟨*file_var*⟩`)`
`rewrite(`⟨*file_var*⟩`,`⟨*file_spec*⟩`)`
`rewrite(`⟨*file_var*⟩`,`⟨*file_spec*⟩`,`⟨*shared_access*⟩`)`

Here, ⟨*file_var*⟩ is the name of a Pascal file variable (for instance, a variable of type `text`, `file of integer`, etc.).

The "⟨*file_spec*⟩" parameter is the file specification. This parameter's type must be a string or a packed array of characters. The ⟨*file_spec*⟩ parameter may include volume specification (such as a volume name, unit number, or SRM directory path) and size specification (such as `[100]` or `[*]`).

The "⟨*shared_access*⟩" parameter is an optional parameter which is used with Shared Resource Manager files to control shared access to the file. See subsequent sections of this chapter called *RESET, REWRITE, OPEN, and APPEND*, *SRM Concurrent File Access*, and *SRM Access Rights*.

Temporary Files

We saw in one of the examples earlier in the chapter that when a new file is first created, it is considered "temporary," and it will remain so until it is closed with a specification that it be saved permanently. Such temporary files don't conflict with other files of the same name. A new file created by `rewrite`, `open`, or `append` will be thrown away when the program terminates *unless* the program takes explicit action.

Size Specification Parameter

The allowable file name syntax depends on the Directory Access Method (DAM) being used; this subject is discussed in a section later in the chapter called *File Naming Conventions*. However, all file names may have appended to them a specification of the size of the file, which the DAM may use at file creation time to allocate space. The size specification may take the following forms.

- Not present. The file will be allocated the largest available block of space for contiguous-file DAMs (LIF and Workstation 1.0 directory organizations), or an indeterminate amount of space for the SRM. Example: `'CHARLIE.TEXT'`.
- `[*]` on the end of the file name. The file will be allocated the greater of these two quantities: 1) the second largest free block, or 2) half of the largest free block for contiguous-file DAMs (LIF and WS1.0); on the SRM, an indeterminate amount of space will be allocated. Example: `SUSANNAH[*]`
- `[`*n*`]` on end of file name, where *n* is a positive integer. The file will be allocated *n* blocks of 512 bytes each for contiguous-file DAMs, or an indeterminate amount by the SRM. Example: `EXACTLY[1000]` is allocated 512 000 bytes.

Anonymous Files

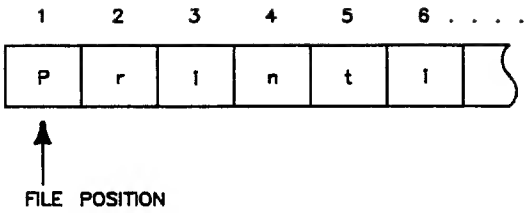
It is permissible to create anonymous files by creating a file without specifying a file name, for example `rewrite(F)`. Such files will always be placed on the system volume. Note however that there is no way to request a specific file size for an anonymous file; `rewrite(F,'[500]')` is not acceptable because there is no file name preceding the size specifier.

The `rewrite`, `open`, and `append` primitives do not necessarily create a new file. Whether they do depends on whether a file already exists with the given name, and whether the file variable is already associated with some physical file by virtue of a previous opening operation.

File Position

In order to understand the three modes a file can be in, we need to take some time to discuss the *file pointer* and the *file buffer*, denoted as the value pointed to by a file variable: `F^`.

A file pointer is associated with each open file. This pointer can be thought of as a marker indicating how much the file has been read or written. For example, the file pointer is initially pointing at the beginning of the file when the file is opened with `reset`, `rewrite`, or `open`. On the other hand, the file pointer is set to the end of the file if the file is opened with `append`. The file element pointed at by the file pointer is called the *current component*. Each time you read from a file, the current component is fetched. Each time you write to a file, the new information becomes the current component.



The components of a file are numbered sequentially from 1 to *n*, where *n* is the number of components in the file. The *file position* is a number from 1 to *n* + 1, which usually corresponds to the position of the file pointer.

The Buffer Variable

Each file has associated with it a special variable called the *buffer variable* or the *file window*. This is a variable of the same type as the components of the file. It is referred to as `F^` where `F` is the file identifier. For example, if `F` is a `file of integer`, then `F^` is an integer variable. The buffer variable is usually associated with the current component of the file.

File States

Every file which is open is in one of three states or modes at any given time depending on what was the most recent operation on that file. The file state has to do with whether you are reading or writing the file and whether you have referenced the buffer variable, `F^`. The three states are as follows:

- *Write* mode
- *Read* mode
- *Lookahead* mode

If the file is in write mode, `F^` has no special meaning other than as a variable, and referencing it causes no I/O to take place. This is the mode in which you normally assign to `F^`; for instance:

```
F^:=⟨data item⟩;
```

in preparation for a `put` statement. If you assign from `F^`; for instance:

```
⟨data item⟩:= F^;
```

in this mode you will get unpredictable results.

The `read` mode is also called the "lazy I/O" state, because in this mode the buffer variable refers to the current component of the file, but the File System does not fill it until the first time it is referenced. In this mode you normally assign from `F^` in order to read the next component of the file.

If the file is in `read` mode, referencing `F^` causes the current component to be fetched from the file and placed in the buffer variable. When this is done, the buffer variable is full and the file goes into the `lookahead` mode. Once the file is in the `lookahead` mode, `F^` may be referenced as many more times as desired but no more I/O will be done.

The lookahead mode is so called because we have "peeked" at the current component without having advanced completely past it. In actuality, the current component has been read into F^ and the file pointer has advanced to the following component. However, the file system pretends that the current component hasn't been fetched yet. In this state the position function returns a value corresponding to the component in the file buffer, which is 1 less than that corresponding to the true file pointer. Also, in this state, read(F,V) will assign the value of F^ to V instead of reading the next component of the file. On the other hand, if a write were done in this state, it would write the component at the true file pointer, and the position function would appear to advance by 2 instead of 1!

REWRITE(F) [with optional 2nd and 3rd parameters]
If F was already open at the time of rewrite and no file name is specified, the same physical file is referenced. If a file name is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit close is actually a close(F,'normal'), and so the file will not necessarily be saved. The file is positioned to its beginning, and any data it contained is discarded. Thus, one way to overwrite the content of an existing file is to open it for reading via reset, then rewrite it.

If the file variable F is not already associated with a physical file (that is, F is not presently open), a new file is created and opened for writing. If a file name and size are specified, they will be applied. The new file created is temporary until it is closed, and in fact is distinct from any existing file of the same name.

OPEN(F) [with optional parameters]
Opens a file for random (direct) access, allowing both reading and writing. The file pointer is positioned to the file's beginning.

If F was already open at the time of open and no file name is specified, the same physical file is referenced. If a file name is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit close is actually a close(F,'normal') and so the file will not necessarily be saved.

If F is not open and no file name is given, an anonymous file is created. If a file name is given matching an existing file, that file is used; otherwise a new file is created.

APPEND(F) [with optional parameters]
If F was already open at the time of append and no file name is specified, the same physical file is referenced. The procedure append positions to the end of the file and re-opens it for writing. If a file name is specified, the current file is closed and the physical file specified by the second parameter is referenced. This implicit close is actually a close(F,'normal'), and so the file will not necessarily be saved. Any data written will get tacked onto the file; the original content remains valid.

If F is not already open and no file name is given, an anonymous file is created and the behavior is like rewrite command.

If F is not open and a file name is given, append searches for an existing file of that name. If one is found, it positions to the end and prepares for writing; if none is found, it creates a new temporary file.

Restrictions on APPEND

Doing an `append` to text files is not allowed in the Series 200 Pascal Workstation implementation. It only works for Data files (`file of` ⟨*type*⟩).

If the file is in a volume with a WS1.0 directory organization, it may not be possible to `append`. For this directory type, `append` is *only* allowed if there happens to be free space on the disc *immediately following* the current end of the file.

Disposing of Files

A program terminates the association between a file variable and a physical file with the `close` procedure. For example, the call may specify that the file is to be deleted from the directory or made permanent. Here are some specific examples:

`close(F,'save');` `close(F,'lock');`	Both do the same thing; the file is made permanent in the volume directory. If the file is anonymous (has no name), then the file is closed and purged. Letter case is ignored.
`close(F);` `close(F,'normal');`	Both do the same thing. If the file is already permanent, it remains in the directory. If it is temporary, it is removed. Letter case is ignored.
`close(F,'purge');`	The file is removed from the directory whether or not it was permanent. Letter case is ignored.
`close(F,'crunch');`	The end-of-file (EOF) marker is set at the current file position; data beyond this position is lost. Otherwise like `'lock'`. Letter case is ignored.

Opening Existing Files

To open an existing file, you must give a file specification to the `open`, `append` or `reset` standard procedures.

RESET(F,'⟨*file_spec*⟩')
Opens an existing file for reading, and positions `F` to the beginning. If `F` was already open and no file name is specified, the file to be read is the one which was open. Otherwise, the file system searches for an existing file of the specified name and reports an error if none is found.

The statement `reset(F)` with no file specification will fail unless `F` is already open.

OPEN(F,'⟨*file_spec*⟩')
APPEND(F,'⟨*file_spec*⟩')
`open` and `append` search for the specified file. If one is found, then the association will be with that physical file. But note that if no file is found, then a new temporary file will be created (see the comments about file creation shown above).

Note that `open(F)` and `append(F)` without a file specification will create new files unless `F` was already open.

REWRITE(F,'⟨*file_spec*⟩')
When `rewrite` specifies the name of a file which already exists, a new temporary file is created. All output data goes to this new file instead of the old one. At the time the file is closed using `close(F,'lock')` or `close(F,'crunch')`, the old one is purged and the temporary file is renamed. Both `close(F,'normal')` and `close(F,'purge')` will purge the new file, leaving the old file intact. This prevents destruction of the old file in case the program terminates prematurely.

To get rid of the old file first, open it with `reset` and then do a `close(F,'purge')`.

Sequential File Operations

In Pascal there are two classes of files: `text`, or line-oriented file, and Data, or item-oriented files. Files of type `text` are so declared in the Pascal program:

```
var
  F:              text;
```

Text, or line-oriented, files are best thought of as lines of characters, separated by end-of-line designators of some sort. They are intended to represent humanly readable text material such as documents.

Data, or item-oriented, files are files of some component type. They are ordered sequences of variables, all of the same type. The type may be a predeclared type like `integer`, or some user-declared type:

```
type
  Rec=          record
                  Name:           string[50];
                  SocialSecurity: integer;
                end;
var
  SS:           file of Rec;
```

A `file of char` is not the same thing as a text file, because no lines are distinguished in the `file of char`.

This section is about Data files; the discussion of text files is below. In the discussion, `F` denotes a file variable; `T` is the type of its components; and `V`, `V1`, `V2`, etc., are variables of type `T`.

READ(F,V)
If `F` is open for reading (by `reset` or `open`), then this standard procedure will store into variable `V` the current component of `F` and advance to the next component. Note that `read(F,V1,V2,V3)` is equivalent to three `read`s in a row. In the `lookahead` mode, `read(F,V)` assigns the value of `F^` to `V` instead of fetching the next component of the file (i.e., no I/O is done).

WRITE(F,V)

If F is open for writing (by rewrite, append, or open), then the value of V is written as the current component of F, and F is advanced to the next component. write(F,V1,V2,V3) is allowed.

The file variable name can be referenced as a pointer. It points to the "current" component of the file; that is, if F is a file of T, then F^ is a variable of type T. F^ is called the "buffer variable" of F. (This logical buffer is distinct from the physical device buffer!)

HP Pascal specifies the use of "lazy evaluation", which simply means that the buffer variable is not filled until the program references it.

PUT(F)

The put and write operations are related. To output data using put, first store into the buffer variable the value to be written, then call put:

```
F^:=V;
put(F);
```

This sequence is equivalent to:

```
write(F,V);
```

Note that it isn't enough to just store into F^; you must also put the value. For instance:

```
F^:=V1;
F^:=V2;
put(F);
```

will store into the file the single value V2. Also, if you fail to put the last component before closing the file, the last component will be lost.

The put(F) operation writes the buffer variable, F^, to the current component of the file. That means that these two statements:

```
F^:=V;
put(F);
```

are equivalent to this statement:

```
write(F,V);
```

GET(F)

This is the complementary operation to put, used for input. It throws away the current component value and advances the file to the next component.

In write mode, get changes the state of the file to read mode, but does not change the file position or do any I/O. For example:

```
open(F,'file_spec');  {puts file in write mode}
get(F);               {puts file in read mode}
V:=F^;                {fetches first file component into V}
```

In `read` mode, `get` causes one component to be fetched from the file, which advances the file position by 1, but that component is discarded. For example:

```
reset(F,'file_spec');  {puts file in read mode}
get(F);                {reads and discards one component}
V:=F^;                 {fetches second file component into V}
```

In `lookahead` mode, `get` discards the component in the file buffer, `F^`, and changes the state of the file to `read` mode. This causes the file position to reflect the true file pointer, thus appearing to advance it by 1. For example, this sequence of statements:

```
reset(F,'filename');   {puts file in read mode}
V:F^;                  {fetches first file component into V}
get(F);                {discards F^, advances position}
```

Direct Access (Random Access) Files

Files of type Data (item-oriented files) may be accessed directly; that is, a program can specify that it wants to read or write the *n*th record in the file without scanning through the records in sequence. A file must be opened with the `open` procedure to allow direct access.

The components of a direct access file are numbered sequentially, with the first being number 1. (Note that there is no acknowledged standard in this area; for instance, UCSD Pascal numbers the first component of a direct access file as record 0. However, all HP Pascal implementations work as described herein.)

When a file is opened, it is positioned at the first component. If sequential I/O operations are performed, the file components will be accessed in ascending order. There are several ways to randomly access the *n*th record.

READDIR(F,N,V)

The read-direct standard procedure positions `F` to component `N` of the file, and then reads the value into variable `V`. Subsequent `read` calls would receive records $n+1$, $n+2$ and so on. `readdir(F,N,V1,V2,V3)` is equivalent to the following sequence:

```
readdir(F,N,V1);
read(F,V2);
read(F,V3);
```

Also:

```
readdir(F,N,V);
```

is equivalent to:

```
seek(F, N);
read(F, N);
```

WRITEDIR(F,N,V)

The write-direct procedure positions `F^` to component *n* of the file, and then writes value `V`. Subsequent writes will place values in components $n+1$, $n+2$ and so on. For example:

```
writedir(F,N,V1,V2,V3);
```

is equivalent to:

```
writedir(F,N,V1);
write(F,V2);
write(F,V3);
```

Also:

```
writedir(F,N,V);
```

is equivalent to:

```
seek(F,N);
write(F,V);
```

SEEK(F,N)

As with the other direct-access procedures, file F must be opened (for both read and write). The procedure seek positions F^ so that the next call to read or write will fetch or place component N.

```
open(F,'CHARLIE');
seek(F,100);
get(F);
V100:=F^;
```

This definition is certainly counter-intuitive in that the program *must not* do an initial get after opening the file, but *must* after the seek command.

The procedure seek works most smoothly (in the most natural fashion) if used with read and write:

```
seek(F,N);
read(F,V);
```

Remember that seek leaves the file in write mode, so that in order to read the current component by referencing F^ you must first do a get command. That means that the following sequence:

```
seek(F,N);
write(F,V);
```

is the same as this sequence:

```
seek(F,N);
F^:=V;
put(F);
```

However, this sequence:

```
seek(F,N);
read(F,V);
```

is equivalent to the following sequence:

```
seek(F,N);
get(F);
V:=F^;
get(F);
```

POSITION(F)

This function returns an integer value which is the number of the next component which will be read or written. If the buffer variable `F^` is full, `position` returns the number of that component.

Please be cautious with this function if the file is in the `lookahead` mode (i.e., if you have read the current component by referencing `F^`). In this mode, `position` is correct for reading, but it is 1 less that the correct value for writing.

MAXPOS(F)

This function returns an integer value which is the number of the last component which has ever been written into the file. Note that the component must have been written; merely `seek`ing out to some far component is not enough to cause the maximum position limit to be extended.

Text Files INPUT and OUTPUT

A text file is composed of variable-length lines of characters. It differs from `file of char` in that the lines are separated by end-of-line marks. As mentioned at the beginning of the chapter, the Pascal Workstation File System supports three different text file representations. Text files are the basis of human-legible input and output. This means that they are used for "formatted" I/O, such as printouts.

Declaring a Text File

A text file must normally be declared in the following way:

```
var
  F:          text;
```

All text files must be declared, except the two standard files `input` (corresponding to keyboard) and `output` (which sends its output to the CRT). These two files, if used, must be listed in the main program header as follows:

```
program X(input,output);
```

However, they must *not* be declared in the body of the program.

In addition, there are two other "standard" system files which may be used, called `keyboard` and `listing`. If these two files are used, they must appear both in the program heading and in a `var` declaration, as follows:

```
program X(input,output,keyboard,listing);
var
  keyboard,listing:  text;
begin
 . . .
end.
```

Don't worry about why `input` and `output` must not be declared yet `keyboard` and `listing` must be; that's how it is. Note also that the four standard files are automatically opened by the Operating System before the program runs. The standard files do not generally appear in `reset` or `rewrite` statements, although they may be closed and re-opened if necessary. Closing and re-opening standard files is not recommended.

The files `keyboard` and `input` both take characters from the keyboard; the difference is that characters read from `input` are echoed to the CRT, while those read from `keyboard` are not. The file `listing` is opened to `PRINTER:listing.ASC` which is the standard system printer. (Note that since `PRINTER:` is normally an unblocked volume, the file name part of the specifier is ignored. On the other hand, if `PRINTER:` is a mass storage volume, the file name is significant. It's a good habit to include a file name even when going to unblocked volumes.)

Representations of a Text File

The way lines of characters will be represented in a text file depends on the *file type*, which is determined when the file is originally created. The three file types are as follows:

- Text (suffix `.TEXT`)
- ASCII (suffix `.ASC`)
- Data (no suffix)

If the file name given in the `rewrite` statement which creates the file ends in the suffix `'.ASC'`, the file representation used is LIF (Logical Interchange Format) ASCII. In this representation, each line is preceded by a signed, 16-bit length field telling how many characters are in the line. In this representation, there is no restriction on what characters may appear in the line. (However, note that ASCII control characters will cause problems with the EDITOR subsystem.)

If the creation file name ends in the suffix `'.TEXT'`, the representation used is known as "Workstation 1.0" (or WS1.0) format. This format is compatible with the UCSD Pascal P-system textfile representation, and may be used as an non-HP interchange format.

The WS1.0 format precedes lines with an optional leading-blank compression indication, and terminates each line with an ASCII carriage-return character. Leading blank compression occurs when a line is written, and the compressed blanks are expanded when the line is read. When using this format, don't write the characters NUL (`chr(0)`), CR (`chr(13)`) or DLE (`chr(16)`). Moreover, note that tabs (`chr(9)`) are not expanded! Generally it is wise to avoid writing any characters with ordinal value less than 32 into WS1.0 textfiles.

If the text file is created anonymously (no file name given) or without a known suffix, the "Data" file representation is chosen. In this case, a carriage-return denotes end-of-line, and all other characters are passed through uninterpreted.

Note

If a file is to be used by the Editor, then you should not store control characters (characters with ordinal values less than 32) in it. These characters may cause erroneous cursor placement, which results in data being inserted or deleted in the file at the wrong place.

Note

The representation of a text file is *not* a function of the directory format being used. A LIF ASCII file may be present in a WS1.0 directory, or a `.TEXT` file in a LIF directory.

The LIF ASCII representation can only be used if the LIF ASCII Access Method module (ASC_AM) is installed in your system's BOOT:INITLIB file. The WS1.0 format can only be used if the UCSD Text Access Method (TEXT_AM) module is installed in INITLIB. These modules are present in INITLIB when the Pascal system is shipped, but can be removed if not needed.

If the required Access Method is not installed, the system will choose the "Data" file representation regardless of file name suffix.

Formatted Input and Output

The use of write, writeln, read, and readln to write formatted output to text files is described in many Pascal reference documents and will not be repeated here, except to take note of the behavior when reading and writing character strings.

HP Pascal supports two forms of character strings, generically referred to as PAC (for packed array [1..*n*] of char) and string. A PAC is a variable whose type specification is of the form

```
type
  T=          packed array [1..n] of char;
```

where *n* is some integer constant. The lower bound of a PAC subscript *must be 1* in HP Pascal, although Series 200 Workstation Pascal allows any arbitrary lower bound if the $UCSD$ Compiler option is used.

When a string literal value is assigned to a PAC, and the string is shorter than the declared PAC length, then the literal string is blank-padded to the declared PAC length before it is placed in the PAC. Thus, if a 5-character literal is assigned to a 10-character PAC, the last 5 characters of the PAC will get blanks. This same behavior occurs on input of a PAC value (see below).

When a PAC is written to a text file, all *n* characters are put out unless a shorter field specification is given in the write statement:

```
type
  PAC=                packed array [1..10] of char;
var
  S:                  PAC;
      .
      .
      .
S:='abcde';           {pad with 5 trailing blanks}
write(F,S);           {write 10 characters}
write(F,S:5);         {write first 5 chars}
write(F,S:15);        {write 5 blanks, then all 10 chars of PAC}
```

A string is a variable whose type specification is of the general form:

```
type
  S=          string[n];
```

where *n* is a constant between 1 and 255 giving the maximum allowable length of the string. Strings differ from PACs in having an implicit variable "current" length. Usually the length of a string is the length of the last string value assigned to it, although string length can be explicitly manipulated by the standard procedure `setstrlen`.

When a string variable is read from a text file, its length is set to the length of the incoming string (see below). When written, a string takes the number of characters specified by its current length.

Reading a STRING or PAC from a Text File

When a string is read from a text file, its length is usually determined by an end-of-line marker.

If the entire string is filled before end-of-line is reached, the read operation ceases, as we saw in the example program earlier. No error is reported, and the next character read will be the one following the last one read.

When reading strings, an end-of-line must be explicitly passed by `readln`. If you repeatedly read into a string while positioned at an end-of-line marker, you will keep getting back an empty string or a PAC of all blanks. The approved way to read long lines into short strings is:

```
while not eof(F) do
  begin
    repeat
    read(F,S);
    ⟨process the piece of string⟩
    until eoln(F);
  readln(F);
  ⟨any other desired processing⟩
  end;
```

You should be aware of one other fact about end-of-line handling in `reads`: reading strings or PACs is the only situation in which end-of-line is not automatically "swallowed". The Standard states that when `eoln(F)` is true, the value of `F^` is a blank. When reading a number, for instance, end-of-line is not treated differently from any other blank in the character stream of the input text file.

RESET, REWRITE, OPEN, and APPEND

The optional third parameter (⟨*shared_access*⟩) to the standard file opening procedures is used at the time of file creation to control concurrent access to files and to specify file access rights via passwords. This parameter is a character string whose syntax conforms to the following definition:

```
shared_access ::= [ concurrency_word ]
              ::= [ password_list ]
              ::= concurrency_word ";" password_list

concurrency_word ::= "SHARED"
                 ::= "EXCLUSIVE"
                 ::= "LOCKABLE"

password_list ::= capability [ ";" capability ]

capability ::= password ":" access_right_list

access_right_list ::= access_right { "," access_right }

access_right ::= "READ"
             ::= "write"
             ::= "PURGELINK"
             ::= "CREATELINK"
             ::= "SEARCH"
             ::= "MANAGER"
             ::= "ALL"
```

Note that in the passwords themselves, uppercase and lowercase letters are distinct. Examples of ⟨*shared_access*⟩ are as follows:

```
'SHARED'
'EXCLUSIVE,MYSECRET:MANAGER'
'LOCKABLE,R:READ;W:WRITE'
'Charley:ALL'
```

Debugging Programs Which Use Files

The File System uses the `try/recover` and `escape` mechanisms (two System Programming extensions) in its normal internal operations. For instance, when opening a file, several escapes may occur internal to the File System or driver calls. However, these "errors" don't get passed on to the user program.

However, if the Debugger is used on such a program and error trapping is enabled, the Debugger will stop the computer on each internal escape. This behavior can be very confusing unless you understand what is happening. The telltale clue that this is happening is that the line number displayed by the Debugger (lower, right corner of the screen) doesn't change during the File System call.

The most common escape codes generated in this fashion are −10, 2080, and −26. You can suppress the Debugger's activity on these codes with the following "Escape Trap Not" Debugger command:

```
ETN -26 2080 -10
```

SRM Concurrent File Access

Three modes of access to shared files are allowed:

EXCLUSIVE — No concurrency. Only one workstation may open the file at one time. This is the default for all files opened on the SRM.

SHARED — No controls. The file may be opened by any number of workstations for both reading and writing. This is particularly dangerous for multiple writers since, for performance reasons, some local buffering is done in each workstation. Different buffers may overlap parts of the same file, and may not contain identical data! Shared file users will not be aware of changes in actual end-of-file induced by the actions of other users.

LOCKABLE — This mode provides for strict concurrency interlocking by means of the `lock`, `waitforlock`, and `unlock` file operations. The file must be locked to perform any operation on it; only one reader/writer may access the file at a time. A series of operations or a single operation may be performed while it is locked. The initial lock obtains the necessary physical file status information from the SRM, and unlocking updates all the information on the SRM as well as flushes its buffers. Thus, when the file is unlocked, its contents are always complete and consistent.

The user-callable routines which support locking are provided in the library module called `lockmodule`, which is in the standard `LIBRARY` file (on the `SYSVOL:` disc). To use them, the program must `import lockmodule`. These specifications for these routines are as follows:

- `function lock (anyvar F: file): boolean;`

  This function returns `true` if the lock succeeded, or `false` if the lock failed because the file was already locked. Other I/O errors, such as `File not open`, generate an error condition which may be trapped by using `try/recover` (see the System Programming Language Extensions section of the *Pascal Language Reference.*)

- `procedure waitforlock(anyvar F: file);`

  This procedure sends the SRM a request to lock the file, and then waits until it is confirmed.

- `procedure unlock(anyvar F: file);`

  This procedure releases the file so that another workstation can lock it.

File locking capabilities are primarily intended for data files (Pascal `file of` ⟨*type*⟩) which are opened for random access using the standard procedure `open`. Suppose that `F` is a file which is not already open. The cases are as follows:

- `open(F,'`⟨*file_spec*⟩`');`

  The existing file is opened for exclusive access. The open will fail if the file is already open by some other workstation. This is the default.

- `open(F,'`⟨*file_spec*⟩`','EXCLUSIVE');`

  The existing file is opened for exclusive access. The `open` will fail if the file is already open by some other workstation. There are three ways to fix this, and they are presented in the order to attempt them: 1) Press [I] (for Initialize) from the main command prompt. This usually closes files opened by your workstation. 2) Rerun the configuration table program (`TABLE`). You can do this either by executing it like any other program ([X] from the main command level), or rebooting. 3) Shut down the workstation activity from the SRM console.

- `open(F,'`⟨*file_spec*⟩`','SHARED');`

  The file is opened for shared access. Any number of workstations may have the file open `SHARED` at the same time. They may read or write–there is no synchronization.

- `open(F,'`⟨*file_spec*⟩`','LOCKABLE');`

  The file is opened in such a way that no access is permitted unless the file is first put in the locked state. Any number of workstations may have a file open `lockable` at a time, but only one workstation may have the file locked.

A `rewrite`, to a file which is already open within the program performing the `rewrite`, simply repositions the file to it beginning and sets it up for writing.

If `rewrite` specifies the name of a file which does not exist, a new file of that name is created and used.

If a physical file name is given and a file of that name exists, the existing file is opened with whatever concurrency specification (SHARED, EXCLUSIVE) was given in the `rewrite`. If no physical file exists, one of the given name is created and opened with the requested concurrency specification. This action is in addition to the creation of the temporary file, and helps prevent interference by other workstations.

Surprising effects may occur if two workstations `rewrite` the same physical file concurrently. The one closed last will remain in the SRM directory.

Note that `rewrite(F,'LOCKABLE');` is probably not a sensible operation. However, it does not generate an error.

SRM Access Rights

Passwords can be used to restrict the types of access allowed to a file (on the SRM, a directory is also a file). They can be set by the Filer's Access command, or at the time that a file is created. Passwords can control the following six types of access:

- READ
- WRITE
- SEARCH
- CREATELINK
- PURGELINK
- MANAGER
- ALL

Any access rights for which no password is specified belong to the set of public capabilities which are granted to any workstation opening the file without specifying passwords.

The word ALL denotes the six access types collectively. When an ALL password exists, there are no public capabilities. The ALL password allows any file operation to be performed.

SEARCH capability is required on all directories along the directory path to a given file.

The reset operation requires READ access to the file.

Both READ and WRITE capability are required if the file is opened by calls to open or append.

To rewrite an existing file, any passwords in the file specification (second parameter to rewrite) are used only to purge the old file. However, one of the three capabilities READ, WRITE, or MANAGER must also be granted to open the file before purging it. The new file created by rewrite will have the passwords specified in the third parameter; until this new file is closed, any operations may be performed on it.

The WRITE capability on the directory in which it resides is required to close-with-'purge' a file, in addition to the SEARCH capability needed to open the file and PURGELINK capability on the file.

To close-with-'lock' a file, WRITE capability is required for the parent directory, in addition to the SEARCH capability needed to open the file.

If a password with MANAGER capability is used to open a file, any file operations may be performed, since the manager password would allow access types to be changed. For example, the following statement gives no public capabilities:

```
rewrite(F,'FILE1','A:ALL');

rewrite(F,'FILE1','M:MANAGER');
```

This statement keeps all capabilities except MANAGER public. This method allows any file operations to be performed, but the manager password 'M' is required to change or set passwords.

How Magnetic Discs Work

Now that the "theoretical" groundwork has been laid and we know how Pascal uses mass storage devices, how do they *really* work? How do bits stick to that little piece of plastic or aluminum?

Discs come in two types: "flexible" and "hard." Flexible discs are also known as "floppy discs" since they are light, thin, and can be bent slightly. Hard discs are sometimes called "fixed," since the disc is not removable from most hard disc drives.

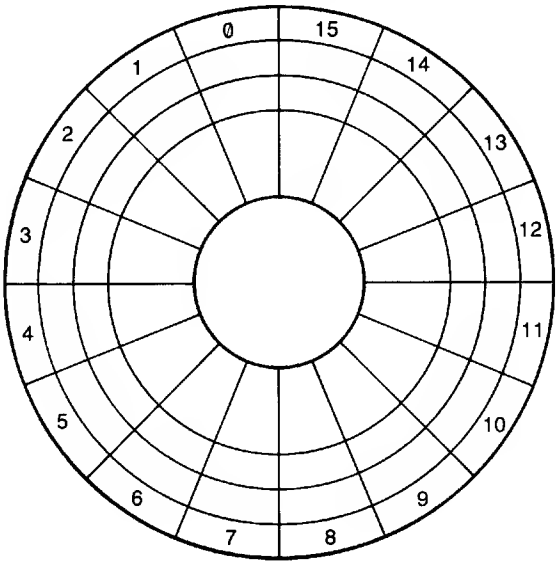
Both types of discs work in essentially the same way. The disc is a platter similar to a phonograph record made of plastic or metal. The disc is coated with a smooth layer of microscopic, magnetizable particles similar to that used in tape recorders. When the disc is in a disc drive, it spins very fast. As it spins, a magnetic sensor similar to the record/playback head in a tape recorder is held over the disc's surface. The disc drive has a mechanism used to move this head over various parts of the disc's surface.

The recording groove in a phonograph record is a continuous spiral from the outer edge to the middle. By contrast, magnetic discs are organized into a sequence of concentric but unconnected circular tracks. The computer must tell the disc drive where to place the head over a particular track in order to read or write data. The tracks themselves are logically broken up into blocks of data called sectors. Discs are often referred to as "blocked devices" because of this structure.

The smallest amount of data that can be read from or written to a disc is a single sector. The computer may read or write several sectors in immediate succession. Since the disc is spinning, the computer must usually wait until the desired sector rotates into position under the head once the recording head is positioned over the correct track. By processing one sector after another as fast as the disc is rotating, the time delay caused by waiting for the sector to get into the correct position can be effectively eliminated.

For various reasons, the computer may, after processing a sector, not be ready for the next one as it spins into position. By staggering the sectors on the disc it is possible to insure that the next logical sector rotates into place just when the computer is ready for it. This staggering technique is called *interleaving*, and it can greatly improve your system's performance. Using the wrong interleave factor can likewise drastically reduce your system's performance.

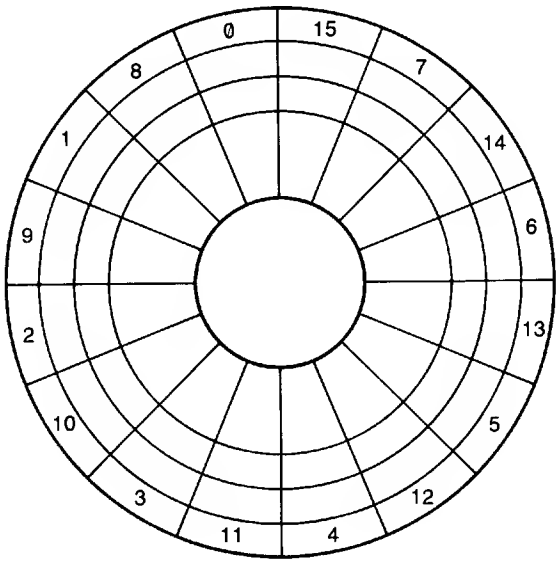
For example, imagine a track that has 16 sectors of data numbered 0 through 15. If the disc has an interleave factor of 1, the sectors are simply accessed in order of occurrence on the disc:



A track of a disc with interleave factor of 1.

After reading sector 0, the computer must immediately be ready for sector 1. If the computer isn't ready for sector 1, it will be missed and sectors 2 through 15 and 0 will pass under the head before sector 1 is again accessible. Thus, only one sector would be read on each disc rotation, not fifteen, which is highly inefficient.

Now suppose the computer's busy period after reading a sector is just a little less than the time that elapses while the next sector passes under the head. By placing sectors out of order on the disc as follows:



A track of a disc with interleave factor of 2.

the computer can access sector 0, skip sector 8, access sector 1, skip 9, and so forth. It is not necessary to wait for an entire disc rotation between each pair of sectors. The numbering scheme shown is said to have interleave 2, since looking at every other sector accesses them in logical sequence.

The interleave factor for flexible discs is established by a process called initializing (some manufacturers use the term "formatting"), which must be done before the disc is used. Initializing is done by a utility program called `MEDIAINIT` supplied with your Pascal system. `MEDIAINIT` knows the appropriate interleave factor to use with various models of disc drives. The default interleave for the disc you are initializing is shown in one of `MEDIAINIT`'s prompts. This default is generally the best interleave factor for that particular device. For example, an HP 8290X defaults to an interleave factor of 3.

For hard discs, the interleave factor is established at the factory, and cannot be changed. Thus, initialization of hard discs serves mainly to find bad tracks and force the use of spare tracks, if necessary, not to change the disc's interleave.

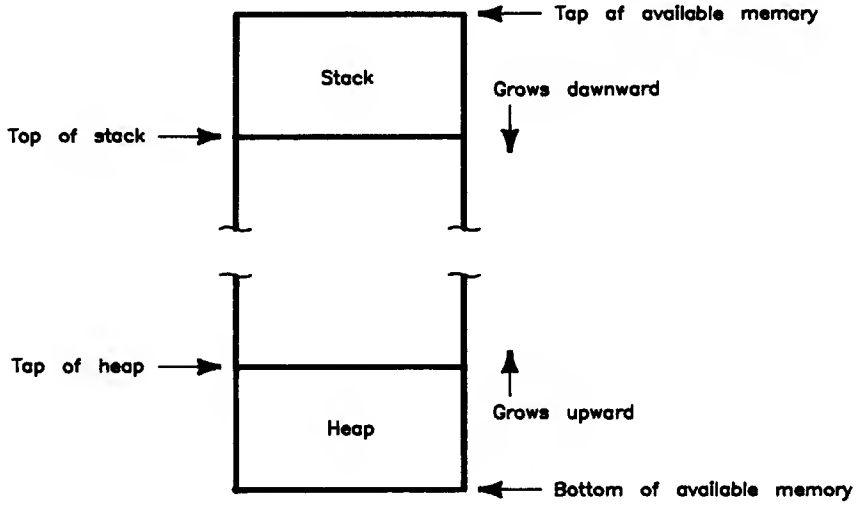
Notes

Chapter 16 Dynamic Variables and Heap Management

Stack/Heap Architecture

The stack and the heap are two data structures inside your Pascal operating system which are used when procedures are called, variables are allocated, etc. The "heap" is the area of memory from which so-called dynamic variables are allocated by the standard procedure "`new`". When a program begins running, it has available one area of memory for data. The program's stack begins at the high-address end of this area and grows downward; the heap begins at the low-address end and grows upward. If the stack and heap collide, a Stack Overflow error (`escapecode` = −2) is reported.

Conceptually, they look like this:



Dynamic Variables and Pointers

In more elementary Pascal programs, most variables are *static* variables; that is, their storage space is allocated at the beginning of the program, and it remains allocated for the duration. This is adequate for many applications, but can cause problems at other times.

For example, when dealing with large arrays, often you do not know how big the array must be. When you run the program, the program may crash because the array is not big enough for this particular run. So, you increase the array size and, the next time, you get a memory overflow error; the machine does not have enough memory to allocate space for the entire array.

One way to deal with this problem is to let the program figure out–while it's running–how many elements it has to deal with. This means that the program allocates memory as processing takes place, and the memory used for one execution of the program is not necessarily the same as for another execution of the program.

Another example of where static variables are insufficient for the task is when your data items are very large–records or arrays of a kilobyte or more apiece–and you want to sort them. If you sort in such a way that you move the kilobyte-sized pieces around, the sort will take much longer than it needs to. The alternate method of just moving pointers is must faster for the machine to carry out, as pointers are four bytes apiece, regardless of the size of the object they point to.

Heap Management

Two disciplines are available for the recovery of the memory used by heap variables after they become unwanted: the `new/dispose` method, and the `mark/release` method. The first is more general; the second is simpler and faster.

We saw the `new` function being used in the example programs earlier in the chapter, but in those applications, the data was being used until the end of the program, so there was no need for the selective removal of individual data items that `dispose` affords.

Calls to `dispose` will have no effect (the freed storage will not be reused) *unless* the main program and the modules containing the `new` and `dispose` calls are compiled with the Compiler option `$heap_dispose on$`.

MARK and RELEASE

This method uses two standard procedures to manage the heap in a purely stack-like fashion. The `mark` procedure is called to set a pointer to the next available byte at the top of the heap. Subsequent calls to `new` will all take space from above this point. When the program finishes with all the variables above the mark, `release` is called to move the top of the heap (the next available space) back to the value saved by `mark`.

```
program markrelease;
type
  ptr = ^ rec;
  rec = record
          f1,f2: integer;
        end;
var
  top,p: ptr;
  i: integer;
begin
  mark(top);        (* remember the base of the heap *)
  repeat
    for i := 1 to 5000 do
      begin
        new(p);     (* allocate from next highest heap address *)
        ...
      end;
    release(top);   (* cut back the heap; recover all space *)
  until false;      (* program will run forever *)
end.
```

When using this method, the computer does not prevent you from making the mistake of releasing to a point **above** the current top-of-heap!

DISPOSE

Alternatively, the standard procedure `dispose` can be used to return each unwanted dynamic variable back to a pool of free space.

Again, calls to `dispose` will have no effect (the freed storage will not be reused) **unless** the main program and the modules containing the `new` and `dispose` calls are compiled with the Compiler option $heap_dispose on$.

```
program disposal;
type
  Ptr=              ^Rec;
  Rec=              record
                      Next:         Ptr;
                      F1, F2:       integer;
                    end;
var
  Top, P, Root:     Ptr;
  I:                integer;
begin
mark(Top);                                  {remember the base of the heap}
  repeat
  Root:=nil;
  for I:=1 to 5000 do
    begin
    new(P);                  {after disposes, will allocate from free list}
    P^.Next:=Root;
    Root:=P;                                {chain all cells together}
    ⟨do whatever other processing is desired⟩
    end;
  ⟨do whatever other processing is desired⟩
    repeat                                  {give back all cells one at a time}
    P:=Root;
    Root:=Root^.Next;                                   {follow the chain}
    dispose(P);                          {memory manager puts on a free list}
    until Root=nil;
  until false;                              {program will run forever}
end.
```

The recycling algorithm takes advantage of the fact that programs which use the heap operate on a great many variables of just a few types. Each type has a characteristic size. When a variable is disposed, it is saved at the front of a list of other variables of the same size. When a variable is allocated, the `new` routine first looks on the list corresponding to the size required; if there is a free object there, it can be allocated immediately. Usually there will be very little computational overhead for either `new` or `dispose`.

The memory manager maintains free lists for objects of sizes 4, 6, 8, through 32 bytes, and one more list for all larger objects. Objects are allocated from this last list on a first-fit basis. No dynamic variable is ever allocated an odd number of bytes.

It is possible for the program to behave so that the heap becomes fragmented (broken into many small pieces). If a request then arrives to allocate space for a large variable, the memory manager will try to recombine the fragments to make a piece big enough to satisfy the request. The fragments must be sorted by address and adjacent ones merged.

The recombination process takes much longer than a simple allocation. Consequently, in real-time applications it is important to analyze the dynamic behavior of programs which use `dispose`.

Mixing DISPOSE and RELEASE

It is also possible to mix the regimes in a well-behaved manner. However, not all implementations of HP Pascal allow mixing these methods in a program. A program which does so may not run properly on other implementations.

If you `release` a properly `mark`ed pointer after some calls to `dispose`, the memory manager will leave on the free lists all disposed objects whose addresses are below the released location. All the space above the released location becomes free, whether or not it was disposed.

During this process the memory manager also recombines any adjacent free fragments, so `release` can also be used to reduce fragmentation. Just `mark` the current top of the heap, then immediately `release` to the same spot.

With the information given in this chapter, you should be more prepared to deal effectively with dynamic variables and their capabilities.

Chapter 17
Error Trapping and Simulation

Introduction

The Systems Programming extensions to HP Series 200 Workstation Pascal have been provided to support error trapping and recovery. In order to use this mechanism, you will need to include the `$SYSPROG$` (or `$SYSPROG ON$`) compiler directive at the beginning of the source program text.

Error Trapping and Simulation

The `try/recover` statement and the standard function `escapecode` have been added to the Pascal language to allow programmatic trapping of errors. The standard procedure `escape` has been added to allow the generation of soft (simulated) errors.

```
try
  ⟨statement⟩
  ⟨statement⟩
      . . .
  ⟨statement⟩
recover
  ⟨single, possibly compound, statement⟩
```

When `try` is executed, certain information about the state of the program is recorded in a marker called the *recover block*, which is pushed on the program's stack. The recover block includes the location of the corresponding `recover` statement, the top of the program stack, and the location of the previous recover block if one is active. The address of the recover block is saved, then the statements following `try` are executed in sequence. If none of them causes an error, the `recover` is reached, its statement is skipped, and the recover block is popped off the stack.

But if an error occurs, the stack is restored to the state indicated by the most recent recover block. Files may be closed, and other cleanup takes place during this process. If the `try` was itself nested within another one, or within procedures called while a `try` was active, that previous recover-block becomes the active one. Then the statement following `recover` is executed. Thus, the nesting of `try` statements is *dynamic*, according to calling sequence, not statically structured like nonlocal `gotos` which can only reach labels declared in containing scopes.

The recovery process does not "undo" the computational effects of statements executed between `try` and the error. The error simply aborts the computation, and the program continues with the `recover` statement.

When an error has been caught, the function `escapecode` can be called to get the number of the error. There are no parameters to `escapecode`. It returns an integer error number selected from the error code table (see the "Error Messages" appendix of the *Pascal Workstation System* manual). System error numbers are always negative.

The programmer can simulate errors by calling the standard procedure `escape(`*n*`)`, which sets the error code to *n* and starts the error sequence. By convention, programmed errors have numbers greater than zero. If an `escape` is not caught by a user recover-block within the program, it will be reported as an error by the operating system. Negative values are reported as standard system error messages, and positive values are reported as a halt code value. Note that `halt(`*n*`)` is exactly the same as `escape(`*n*`)`.

`Try/recover` statements are usually structured in the following fashion:

```
try
  ⟨some operation⟩
recover
  if escapecode=⟨whatever you want to catch⟩ then
     ⟨recovery operation⟩
   else
     escape(escapecode);
```

This has the effect of ensuring that errors you *don't* want to handle get passed on out to the next recover-block, and possibly eventually all the way out to the system. All programs which are executed are first surrounded by the Command Interpreter with a `try/recover` sequence. The recovery action for the system is to display an error message.

The IORESULT Function

Normally the Compiler emits instructions after each I/O statement to verify that the transaction completed properly. If it fails, the program is terminated with an error report.

It is possible to trap I/O errors programmatically, using the `try/recover` statement. The System Programming function `ioresult` can then be called to discover what went wrong with the transaction.

Both the `escapecode` function and the `ioresult` function are needed for the following problem. Suppose, for example, you want to be able to enter values *of an enumerated type* into a program. This is easily done in HP Pascal, but if there is a misspelling, or an invalid token entered, the program bombs on "`error -10: bad input format`". How can this be avoided?

Put the `read` statement in the `try` section, and an error message in the `recover` section. If an error occurs during the read operation, `escapecode` and `ioresult` are checked. If they indicate that an illegal token was entered, print an appropriate error message, and ask for the same input again. Put the whole thing in a `repeat/until` loop so it continues until a correct answer is given.

```
$syspro$
program TryRecover(input, output);
var
  Color:        (xxx, Red, Orange, Yellow, Green, Blue, Indigo, Violet);
  IOerror:      integer;
begin
Color:=xxx;     {just a place holder, to see if a valid color was specified}
repeat
  try
    writeln('Enter a color of the spectrum, then press RETURN or ENTER: ');
    readln(Color);
  recover
    begin
    IOerror:=ioresult;
    reset(input);       {clear the file system flags}
    if escapecode=-10 then
      if IOerror=14 {bad input format} then
          writeln('  (color invalid or misspelled)')
      else
        writeln('Escape code: ',escapecode:0,'  ioresult: ',IOerror:0)
    else
      if escapecode=-20 then
        escape(escapecode)
      else
        writeln('Escape code: ',escapecode:0);
    end;
until Color<>xxx;
writeln('You specified "',Color,'",');
end.
```

$IOCHECK$ and IORESULT

Normally the Compiler emits instructions after each file system I/O transaction to verify that the transaction completed properly. If it didn't, the program is terminated with an error report. The error code for all file system I/O errors is −10.

You may wish to intercept I/O errors programmatically rather than have them terminate the program. This can be done two different ways. The program or module must be compiled with the `$sysprog$` or `$ucsd$` Compiler option at the front of the source text. Both these options make available a system programming function called `ioresult`, which returns an integer value reporting on the success of the most recent I/O transaction. A result of zero indicates a successful transaction; other values are given in the "Error Messages" appendix.

Method 1

This method is the preferred one, and is the one used in the previous example. Compile the program or module with `$sysprog$` enabled, and use the `try/recover` statement to trap the errors.

```
$sysprog$
program TrapMethod1(input, output);
var
  Name:          string[80];
  F:             text;
  IOerror:       integer;
begin
  repeat
  write('Open what ',TEXT' file ? ');
  readln(Name);
  try
    reset(F,Name+',TEXT');
    IOerror:=0;                  {If we get here, the RESET didn't fail.}
    writeln('  File successfully opened.');
  recover
    if escapecode=-10 then       {It's an I/O System error.}
      begin
      IOerror:=ioresult;         {Save it (IORESULT affected by WRITELN).}
      writeln('  Can''t open that file.  IOresult: ',IOerror:0);
      end
    else
      escape(escapecode);        {Pass non-I/O errors back to system.}
  until IOerror=0;
end.
```

Method 2
This method is used in UCSD Pascal programs. In order for it to work properly, you must also suppress the error checks normally emitted by the Compiler.

```
$ucsd$
program UCSD_TrapMethod(input,output);
var
  Name:          string[80];
  F:             text;
  IOerror:       integer;
begin
  repeat
  write('Open what ".TEXT" file ? ');
  readln(Name);
  $iocheck off$
  reset(F,Name+'.TEXT');
  $iocheck on$
  IOerror:=ioresult;        {Save it (IORESULT affected by WRITELN).}
  if IOerror=0 then
    writeln('  File successfully opened.')
  else
    writeln('  Can''t open that file.  IOresult: ',IOerror:0);
  until IOerror=0;
end.
```

Note that `$iocheck off$` before the `reset` statement inhibits escape during the statement. However, `ioresult` will still be set correctly.

Extended Error Information

There are three types of run-time errors where your error-trapping will require the examination of extended error information. They are:

- I/O errors (`escapecode` = −10), and
- I/O library errors (`escapecode` = −26), and
- DGL (graphics) errors (`escapecode` = −27).

These are different than other, simpler, run-time errors, in that two values need to be checked in order to ascertain the error that occurred. This is different than, for example, an integer overflow error. In this case, `escapecode` = −4, and the error listings in the back of the *Workstation System Manual* states that −4 means "Integer overflow".

Get the "extended" error information in the following way. A "file not found" error results in escapecode = −10. However, an escapecode of −10 does not indicate *by itself* that some file was not found. The value of −10 only says, "Go look at ioresult for the rest of the definition of the error." Looking at the ioresult tells you that a file was not found. (Accessing the ioresult function requires either $sysprog$ or $ucsd$.)

Similarly, for I/O library errors, you need to check two different places. When you get an error escapecode = −26, all that tells you is that some I/O library error occurred. Now you need to check the I/O library's counterpart to I/O's ioresult, called ioe_error. (By the way, ioe_error and graphicserror, in the next section, are variables, unlike ioresult. Therefore, you do not need $sysprog$ or $ucsd$ to access them.) The value of ioe_error tells you what kind of error occurred. In addition to this, there is a function called "ioerror_message" (imported from iodeclarations) which converts an integer to an appropriate error message:

```
writeln(ioerror_message(ioe_result));
```

Similarly, for graphics errors you need to check two different places. When you get an error escapecode = −27, all that tells you is that some graphics error occurred. Now you need to check the graphics counterpart to I/O's ioresult, called graphicserror. The value of graphicserror tells you what kind of graphics error occurred.

Determining a File's Existence

This section contains a program segment which is both a commonly needed capability and an instructive example. In many software packs, the user is allowed to store some kind of data, often specifying his own file names. Two things can happen at this point:

- A file by the specified name does not exist. Fine; create the file, store the data, and go on.
- A file by the specified name *does* exist. The computer should not automatically erase the old file and create the new one; there might be some valuable data lost. The computer should give the user the option of deleting the old file by that name, or specifying another name for the new file. At this point, another question must be asked; basically: "That file already exists; should I purge it?" If the user says yes, purge the file, create the new one, and go on. If the user says no, ask him for another file name.

Note that the second option above can happen repeatedly. That is, a user, upon being told that a file by that name already exists, can give another file name which already exists. Thus, the routine should repeat infinitely, if necessary, until a satisfactory file name is given.

```
var
  MyFile:       text;
  MyFileName:   string[80];
  Answer:       char;
  IOerror:      integer;
  NoFile:       boolean;
    .
    .
    .
  repeat
  write('File name to create: ');
  readln(MyFileName);
  NoFile:=true;
  try
    reset(MyFile, MyFileName);    {can't use REWRITE here}
    close(MyFile);
      repeat
      write('File "',MyFileName,'" already exists; shall I purge it? ');
      read(Answer);
      writeln;
      if not (Answer in ['n','N','y','Y']) then
        writeln('Please answer "Y" or "N".');
      until Answer in ['n','N','y','Y'];
    if Answer in ['y','Y'] then
      begin
      rewrite(MyFile,MyFileName);
      close(MyFile,'purge');
      writeln('Old file "',MyFileName,'" purged.');
      NoFile:=true;
      end
    else
      NoFile:=false;
  recover
    if (escapecode=-10) and (ioresult=10) then
      {do nothing; we've determined that the file is not found}
    else
      begin
      IOerror:=ioresult;
      writeln('escapecode: ',escapecode:0,'   ioresult: ',IOerror:0);
      halt;
    end;
  until NoFile;
rewrite(MyFile,MyFileName);
close(MyFile,'save');
writeln('File "',MyFileName,'" created.');
⟨continue processing⟩
```

Note that `$sysprog$` must be activated in order to use `try/recover`.

Error Simulation

Here are two different facets to error simulation:

- Having your own set of errors, peculiar to a particular software package. For example, errors 1000 through 1050. These do not interfere or intermingle with any Pascal system errors, so when certain illegal operations in your software pack are attempted, you can cause one of your own errors to happen:

```
program MyProgram;
  . . .
⟨some error condition is detected⟩
halt(1000);
  . . .
```

or

```
$sysprog$
program MyProgram;
  . . .
⟨some error condition is detected⟩
escape(1000);
  . . .
```

(Again, `halt` and `escape` are the same thing.)

- The second facet of error simulation is: you don't *really* have an error, but you want the computer to temporarily think so, in order for it to take appropriate action. For example, suppose you set some conditions in the course of a program. If an error occurs during the condition-setting, you want to put things back in order. If an error doesn't occur, you want to do some processing, and *then* put things back in order. The point is: *either way*, you want to do the same return-to-normal code.

  Using `escape(0)`, you can cause a `recover` block to be entered, but the "error" number, 0, means "no error."

```
try
  ⟨attempt something which, if failure, goes to recover block⟩
  ⟨do processing⟩
  escape(0);  {cause control to go to the RECOVER block}
recover
  ⟨put things back in order⟩
```

  Note that the `escape(0)` causes control to enter the `recover` block in a nice, controlled manner.

Chapter 18 Special Configurations

Introduction

Workstation Pascal System is self-configuring. As it boots, interface/device driver modules in the Initialization Library (BOOT:INITLIB or BOOT2:INITLIB) are loaded into memory and initialized. Then, the TABLE program determines what peripheral devices are connected to the computer (such as local and remote mass storage devices, printers, and so forth); if the driver module(s) for a particular interface or device are in memory, then the TABLE program can usually assign to it a logical unit number which makes it accessible to the File System.

The term "standard configuration" is defined to be any combination of computer and peripheral devices that will be configured by the Pascal system as it is shipped. This chapter describes how to change this "standard" system configuration.

Chapter Organization

This chapter contains many sections; however, they can be essentially split into three categories.

- A description of how the system boots and auto-configures itself.
- Brief descriptions of several possible configurations.
- Procedures for making changes to the "standard" configuration.

The System Booting Process

You will probably want to read about how the system boots and auto-configures itself, regardless of whether you want to change your system's configuration.

Example Special Configurations

Next, you will probably want to scan the possible "non-standard" ways that you can configure your system. The following sections briefly describe several common configurations:

- Changing hard-disc volume sizes
- Setting up several bootable system configurations
- Adding interfaces and peripheral devices
- Setting up the SRM System
- Changing system printers
- Adding Bubble and EPROM cards
- Using alternate directory access methods (DAMs)

Modifying the Configuration

Then, when you know which configuration change(s) you want and which of the procedures you will need to use to make the changes, you can follow the procedures in the third major section of the chapter. These procedures are as follows:

- Coalescing logical volumes on hard discs into larger volumes
- Copying system files and changing their names
- Making an AUTOSTART or AUTOKEYS stream file
- Adding driver modules to INITLIB
- Modifying the auto-configuration program (CTABLE)
- An example SRM configuration

As an example of the first category, suppose you want to connect one HP 9133V Hard Disc Drive and one HP 7912 Disc Drive to your workstation. The standard TABLE program assumes that it should assign 4 unit numbers to the 9133V hard-disc drive and 30 to the 7912. However, since it reserves only 30 unit numbers for *all* hard-disc volumes, the standard TABLE will not be able to access all 34 volumes (if that is the way that these discs have been or will be partitioned); it will either recognize all 4 volumes on the 9133V and only the first 26 on the 7912, or all 30 on the 7912 and none on the 9133V. Probably the easiest way to make all parts of both discs accessible is as follows: first, "coalesce" the *last* 5 logical volumes on the 7912 into one larger volume (to change the number of logical volumes to 26); second, set up the hardware so that the 9133V gets the lower unit numbers (11-14) and the 7912 gets higher numbers (15-40). The alternative way to make all parts of both discs accessible is to modify the the standard TABLE program; the source program is called CTABLE.TEXT, and it is supplied to you on the CONFIG: disc.

An example of the second category was given in the *Pascal User's Guide*. The system files (such as EDITOR, FILER, and so forth) were copied to a hard disc (913x family). No file names were changed. An example AUTOSTART file (the third category) was given in the same guide. It P-loaded some system files.

As an example of the fourth category, suppose you want to use an HP 98625 High-Speed Disc interface and an HP 98620 DMA Controller card with a CS80 disc drive. First, add module DISC_INTF to the INITLIB file (modules DMA and CS80 are in the INITLIB file supplied with your system). Then when the system is subsequently booted, the standard TABLE program will, barring other restrictions, automatically recognize the disc and make it accessible. (An alternate but less "permanent" way would be to eXecute module DISC_INTF after booting the system and then eXecute TABLE again.) You will then probably want to copy most system files to the CS80 disc, which is another example of the first category.

As an example of the fifth category, suppose you want to connect two HP 7912 Disc Drives to your workstation. The standard TABLE program will not make both drives accessible, since it assumes that each disc needs to be allocated 30 unit numbers and assigns all 30 units available for hard discs to the 7912 with the highest priority. In order to access both drives with the File System, you will need to modify the standard TABLE program ("coalescing" will not work in this case). In this type of situation, you may want to change the default number of logical volumes that the system creates on each drive. After re-compiling and then running the properly modified program, the system will recognize and allow you to access all parts of each drive. You will probably want to replace the original TABLE program with the new version so that this configuration will automatically be made at the next power-up and system boot time.

The Booting Process

This section explains what is going on within the machine as the Pascal System is loaded. It is intended to give you a few more insights into how the system works. It does not, however, describe *how* to boot your Pascal system; that topic is covered in the *Pascal User's Guide* provided with your computer. Neither does it describe modules; that topic is covered in the Compiler chapter.

The Boot ROM

Inside the computer is a ROM (Read-Only Memory) that contains the information needed to begin loading an operating system. The loading process is often called "booting" because it is the computer's way of "pulling itself up by its own bootstraps." This ROM is therefore called the "Boot ROM". The Boot ROM is a non-volatile storage device; its contents are not lost when power is removed.

There are currently several different versions of the boot ROM. Thus, the booting process is slightly different depending on which version of boot ROM is in your computer. However, all perform the general steps outlined in this section.

When you power-up, the computer's central processing unit (CPU, which is a 68000-family processor) reads the first few bytes of this ROM, which begins at address 0. These bytes contain such information as the address of the first executable machine-language instruction and initial value of the stack pointer. After loading these values, the processor continues executing routines in the Boot ROM.

The processor next executes routines that perform a self-test and then displays the amount of memory installed in the computer. You may not see the amount of memory displayed if the CRT is just warming up. After self-test, the processor executes another Boot ROM routine that searches various mass storage devices (such as disc drives) for an operating system; the Boot ROM recognizes files of type "Systm" and with name beginning with the letters "SYSTEM_" (or "SYS" with Boot ROM 3.0 and later) as being operating systems. It also searches system ROM for ROM-based systems (such as BASIC).

Depending on its version and how many systems it finds, the Boot ROM will either choose a system or let you choose one (for instance, Boot ROM 3.0 and later versions allow you to choose one if you intervene in the boot process). With Pascal, this "Systm" type file is called "SYSTEM_P" and it will be discussed momentarily.

The Pascal System Discs

The Pascal system is delivered on either 5.25-inch (mini-floppy) or 3.5-inch (micro-floppy) flexible discs. The discs contain the operating system, subsystems like the Editor and Compiler, and several libraries and utility programs. The discs that you received are listed in the *Pascal User's Guide*. You will have to boot Pascal from these discs at each power-up unless you reconfigure your system. The disc called BOOT: contains the SYSTEM_P file that will be loaded into memory first.

The System Boot File (SYSTEM_P)

The BOOT:SYSTEM_P program is an absolute-addressed program that contains the bare minimum Pascal operating system "kernel." (It was created using the Librarian's Boot command.) It is absolute-addressed so that the Boot ROM can use a simple loading routine.

The SYSTEM_P file consists of a linking loader (more elaborate than the loader found in the Boot ROM) and a few support routines. This kernel is loaded into volatile read/write memory (also called random-access memory, or RAM) from non-volatile memory (usually discs). The linking loader then loads the rest of the system.

In this system, there is no "kernel" in the closed sense of the term, such as a closed system like HP-UX. The system has an open design which allows modules to be added to the system – while the system is running. However, the term "kernel" will still be used in this text to describe the minimum working environment.

The loader then continues by completing construction of the operating system by loading the "Initialization Library" called INITLIB, which is also on the BOOT: disc.

The Initialization Library (INITLIB)

This BOOT: library file consists of modules that complete the kernel of the Pascal operating system. (Some of the modules are programs.) These modules mainly provide access methods (or device "drivers") for internal interfaces and peripheral devices.

Installing INITLIB Modules

As each INITLIB module is loaded into memory, it is bound to the operating system by a linking process. After the loading is complete, each program is executed once. The programs in INITLIB are referred to as "installation code;" their purpose is to properly initialize variables or allocate storage that will be used by these modules. Many interface-driver modules check to see if the interface they are to drive is there, and if not they don't install themselves.

Once INITLIB is loaded and the installation code has been executed, the system has found and identified all interface cards installed in the machine; however, no scan has been made for peripheral devices.

Auto-configuration of a peripheral device requires the device's driver(s) to be in memory at the time that the TABLE program is run (TABLE will be discussed in the next section). The HPIB module is an example of a driver for HP-IB interfaces. If the driver is part of the INITLIB file, then the device can be interrogated at a later time by TABLE (unless other conditions restrict). If the driver is not in INITLIB, then you must add it to the file (or alternately load the driver into memory by executing the installation program that contains the driver module).

Adding and Removing INITLIB Modules

Since the operating system is an "open kernel," you can add, replace, or delete modules within this library; more details regarding these operations are described in the Adding Modules to INITLIB section of this chapter. You must not change the order of modules in this library; neither should you link them together (with the Librarian), which would result in rendering the programs non-executable.

Module LAST

The last piece of installation code in INITLIB is the program named LAST, which attempts to execute the BOOT: files named STARTUP and TABLE. Here is the algorithm used to load and execute these two files; each file's function is described in a subsequent section.

1. If STARTUP is found on the "Boot volume" (i.e., the same volume on which the Systm file, such as SYSTEM_P, was found), then that program is loaded (but not executed).
2. LAST then looks for TABLE on the Boot volume. If TABLE is found there, then it is loaded and executed; it makes File System volumes accessible. If TABLE is not found, then only the keyboard, screen, and Boot volume will be accessible (see the brief description in the subsequent section called Failure of the TABLE Program).
3. If STARTUP was not found on the Boot volume, then the Boot ROM looks for it on the current system volume (at this point it might not be the Boot volume, because TABLE may have re-defined it).
4. STARTUP is then executed.

The Command Interpreter (STARTUP)

With the Pascal system delivered to you, the BOOT:STARTUP file is the Command Interpreter (or Main Command Level) program. However, you can write any program, optionally Link it with the Librarian, name it STARTUP, and with it replace the existing STARTUP file. It will then be loaded at power-up, instead of the Command Interpreter program.

If you use your own STARTUP program, be careful not to destroy the original STARTUP program. The recommended method is to use the Filer's Filecopy command to make a copy of the BOOT: disc on a blank initialized disc and then replace the STARTUP program with your new STARTUP program on that disc. Use the disc with the new STARTUP to boot the computer and your program will start running instead of the Pascal operating system.

The Auto-Configuration Program (TABLE)

The purpose of the TABLE program is to make devices accessible to the File System. Since this is one of the principle topics of this chapter, the subsequent section called Auto-Configuration is devoted to the intricate details of how this program works. For now, let's assume that it has already chosen the system volume and finish this overview of how the system boots.

The AUTOSTART and AUTOKEYS Stream Files

If present on the system volume and if data can be written on the volume (i.e., it is not a read-only volume), the AUTOSTART file is automatically streamed by the system at power-up; if the volume does not permit write operations (such as EPROM cards), then the AUTOKEYS file is streamed, if present. These files must be "stream" files, which are sequences of characters that are used by the system just as if they were commands typed from the keyboard (a "command stream"). Stream files are formally described in the "Main Command Level" chapter.

The AUTOSTART or AUTOKEYS file must be located on the volume designated as the system volume at the point that the TABLE program has finished execution. There is an AUTOSTART file on the BOOT: or BOOT2: disc. Here are the contents of the AUTOSTART file provided with your system.

```
1MAR0

xSWVOL
SYSVOL
3
wsSYSVOL:
qv
```

If you use the original BOOT: disc on a single drive system, it is the AUTOSTART file which causes you to be instructed to place SYSVOL: in the drive and then press the [X] key. This AUTOSTART file then changes the system volume to "SYSVOL:". But because BOOT: is initially the system volume, the AUTOSTART file is found and executed. On dual-drive systems, the media in the second disc (nominally SYSVOL:) will usually become the system volume.

Libraries

The Pascal system is shipped with many library modules. Some are device drivers (in INITLIB or on the CONFIG: disc), while others provide procedures, etc. for applications such as device I/O and graphics (on the SYSVOL:, LIB:, and FLTLIB: discs). Once the system has booted successfully, you can use these libraries. INITLIB modules are described in the Adding Modules to INITLIB section of this chapter. Application libraries are fully discussed in the *Pascal Procedure Library* and *Pascal Graphics Techniques* manuals. You can also write your own libraries, as described in the description of Modules in the Compiler chapter.

The Auto-Configuration Process

A device is only accessible to the File System if it has been assigned a logical unit number. You may be familiar with the Filer's Volumes command, which shows the correspondence between logical unit numbers and volumes. Here is a typical display:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 # BOOT:
  4 * SYSVOL:
  6   PRINTER:
Prefix is - BOOT:
```

The Unit Table

To make devices accessible to the File System, TABLE fills in entries of the Unit Table so as to correctly associate logical unit numbers with logical volumes (and the software required to access the devices on which those volumes exist). The Unit Table is actually a global system pointer variable called "Unitable," which points to a table that contains 50 entries – one for each logical unit (and potential volume). The Unit Table variable is accessed by many parts of the system, such as the Filer, Editor, and Compiler, when they want to use one of the devices.

This section describes how the standard TABLE program assigns Unit Table entries. To see the exact algorithms implemented in Pascal code, refer to the TABLE program called CTABLE.TEXT and corresponding commentary later in this chapter.

Standard Auto-Configuration

The results of a typical auto-configuration process performed by the standard TABLE program are shown in the following table. Each entry is further discussed in subsequent text:

Standard Unit Table

Unit	Nominal Assignment
1	System CRT Screen (CONSOLE:)
2	System Keyboard (SYSTERM:)
3	1st priority floppy (drive 0, primary DAM)
4	1st priority floppy (drive 1, primary DAM)
5	Shared Resource Manager (remote mass storage)
6	System Printer (PRINTER:)
7	2nd priority floppy (drive 0, primary DAM)
8	2nd priority floppy (drive 1, primary DAM)
9	3rd priority floppy (drive 0, primary DAM)
10	3rd priority floppy (drive 1, primary DAM)
11-40	Hard discs (highest to lowest priority)
41	1st priority cartridge tape (LIF DAM)
42	2nd priority cartridge tape (LIF DAM)
43,44	1st priority floppy (same volume as 3 & 4, but alternate DAM)
45	SRM system volume, if appropriate
47,48	2nd priority floppy (alternate DAM for 7 and 8)
49,50	3rd priority floppy (alternate DAM for 9 and 10)

How Unit Numbers Are Assigned

In the Unit Table, certain unit numbers are preferentially assigned to particular classes of devices. Here are the general classes of devices:

- Unblocked devices (i.e., "byte stream" devices that do not have directories) like the keyboard, screen, and local printers
- Floppy disc drives (including 5.25-inch, 3.5-inch, and 8-inch)
- Hard disc drives
- SRM systems
- DC600 cartridge tape drives

The floppy and hard disc drives and tape drives are all "blocked" devices.

Unblocked Devices

To fill the Unit Table, the TABLE program assumes that "unblocked" devices, such as the screen (CONSOLE:), the keyboard (SYSTERM:), and system printer (PRINTER:) are always present and assigns them to units #1, #2, and #6, respectively. However, it must scan for the presence of "blocked" devices (i.e., mass storage devices with directories). Once these devices are found, their locations (select code, HP-IB address, etc.) and attributes (type of disc drive, capacity, etc.) are put in the table entry corresponding to the logical unit number.

Blocked Devices

Here are the steps that the standard TABLE program goes through in assigning unit numbers to blocked devices.

Interfaces and Devices Scanned

In order to find mass storage (blocked) devices, the TABLE program first scans interface select codes 7, 8, and 14 for the presence of an HP-IB type interface: select code 7 is the built-in HP-IB interface; select code 8 is the factory default setting for optional HP-IB interfaces; 14 is the factory default setting for HP 98625 High-Speed Disc interface (a fast HP-IB interface).

If an HP-IB interface is found, addresses 0 thru 7 are interrogated for the presence of blocked devices. (Most HP-IB peripherals identify themselves when asked politely.) The purpose of this interrogation is to determine what type of device (such as what family of disc drive, capacity of drive, etc.) is present at each location.

Device Classes and Unit Numbers
The TABLE program makes a list of the devices found in each of these classes:

- Floppy discs – this class includes all 5.25-inch, 3.5-inch, and 8-inch floppy disc drives, and all CS80 or SS80 devices that have a single *physical* volume with capacity less than 10 Megabytes.
- Hard discs – this class includes all 913x hard discs, and all CS80 or SS80 devices that have either multiple *physical* volumes or a single *physical* volume with capacity greater than or equal to 10 Megabytes.
- Tape drives – this class includes all DC600 cartridge tape drives, such as the HP 9144 Tape Drive as well as tape drives integrated into the CS80 Disc/Tape Drives.

Up to 10 devices can be on the list for each class. If more than 10 devices are found in a class, then only the *last* 10 found are maintained in the list.

As shown in the preceding Standard Unit Table diagram, groups of unit numbers have been reserved for each particular class of devices. For instance, unit numbers 3 and 4, and 7 through 10 are reserved for floppy discs. Unit numbers 11 through 40 are reserved for hard discs. Unit numbers 41 and 42 are reserved for tape drives.

Device Priority
The "priority" of a device is generally as follows: the later in the scanning sequence a device is found, the higher its priority is. Remember that interfaces are scanned in the order of select codes 7, 8 and 14; and on each HP-IB interface, addresses 0 through 7 are interrogated. Thus, a device at 702 has higher priority than a device at 700 but lower priority than one at 800. However, if a device was used to boot the system, then that device will have the highest priority in its category.

Assigning Unit Numbers to Floppy Disc Drives
Units are assigned to floppy discs in pairs, according to device priority. For instance, if two dual-drive floppies are found, then the highest priority floppy device will be assigned unit numbers 3 and 4 and the lower priority device assigned unit numbers 7 and 8. However, if two single-drive floppy devices are found, then the highest priority device will be assigned unit 3 and the lower priority device assigned unit number 7. Up to three floppy drives (and thus pairs of floppy volumes) can be assigned unit numbers.

Assigning Unit Numbers to Hard Disc Volumes
Hard discs are also assigned unit numbers according to device priority; however, there is also another consideration. Since all hard discs currently supported by this system have capacities of several millions of bytes, the standard TABLE prefers to "partition" the *physical* volumes into smaller *logical* volumes. (Some hard discs are also organized to be accessed as four separate physical volumes, rather than one large physical volume; see the subsequent Volume Sizes table for further information).

TABLE sets up Unit Table entries for hard discs according to two factors: the priority of each device, and the number of logical volumes it is assumed to have. Here are the number of units that the standard TABLE will reserve for each hard disc drive, and the corresponding size of each volume.

Standard Hard Discs Volume Sizes

Product Number	Number of Volumes	Volume Size (in bytes)	Volume Size (in sectors)
913x[1] A and V (**Not** Option 10)[2]	4	1 152 000	4 500 (all are same size)
913x A or V (Option 010)[3]	4	1 206 272	4 712 (all are same size)
913x B	9	1 071 360	4 185 (except last = 4340)
913x XV	14	1 031 680	4 030 (except last = 4340)
7908	16	1 030 400	4 025 (except last = 4375)
7911	27	1 032 192	4 032 (except last = 4992)
7912	30	2 179 072	8 512 (except last = 9408)
7914	30	4 390 912	17 152 (except last = 18688)
7933 and 7935	30	13 471 744	52 624 (except last = 53820)

The logical partitioning of hard discs is made by the standard TABLE with the following algorithm. For each device on the list (of up to 10 devices), it calculates the number of volumes required by the device, assuming that the disc is now or will be partitioned; the default number of logical volumes assumed to be on each disc and the size of each volume are shown above. It then begins assigning unit numbers according to device priority; each device is assigned unit numbers according to the number of logical volumes *assumed* to be on the device, regardless of the number of volumes actually on that device. TABLE begins with 11, and continues either until all volumes have been assigned numbers or unit number 40 is reached, whichever occurs first.

At the point that it assigns unit numbers to a device, TABLE has *not* yet determined whether the disc has actually been partitioned. In fact, the disc may not have been initialized yet, or it may have been initialized but not partitioned as assumed. In the second stage of the assignment algorithm, TABLE looks on the disc for each volume's directory. Since these are logical volumes, each directory is assumed to be at an "offset" from the beginning of the disc.

If a valid directory is found at the expected location on the disc (i.e., at the assumed offset), then the corresponding unit number is assigned to the volume. For instance, if a valid directory is found in the first location, then it is assigned the first unit number for that disc (e.g., unit #11 will be assigned to the first directory on the highest priority hard disc device). As each subsequent directory is found, it is assigned the corresponding unit number. For example, if the only hard disc in a system is an HP 9133XV Hard Disc which has been partitioned and initialized according to the standard TABLE volume sizes for this disc, then it will be assigned 14 unit numbers (11-24).

[1] The "x" used here signifies either 9133, 9134, or 9135 products.

[2] The 913x A and V drives (**Not** Option 10) look like 4 separate devices, because they are accessed as *4 separate* "disc units" or "drive numbers."

[3] The 913x A and V Option 10 drives are like the B and XV suffix drives; they are accessed as *1 single* "disc unit" or "drive number."

If a subsequent directory is not found at its expected offset, then that area of the disc is assumed to be part of the last valid directory that preceded this one. For instance, if valid directories are found only in the 1st and 4th expected directory locations on an HP 9133V Hard Disc (assumed to have 4 volumes), then the first volume is assumed to be a coalition of the first three volumes (of the default size) on the disc.

If the first directory is the only valid one found, then the disc is assumed to be one single logical volume. For instance, if the only hard disc in a system is an HP 7911 Hard Disc which has been initialized and partitioned according to the standard TABLE volume sizes for this disc, then it will be assigned 27 unit numbers (11-37). However, if the disc was initialized by the Series 200/300 BASIC system (or coalesced into one volume using the procedure shown later in this chapter), then it will appear as one single, large volume and assigned only unit number 11. In this case, the last 26 unit numbers allocated for the device (12-37) are *not* usable. If another hard disc (with lower priority) were added to this hypothetical system, then it would be assigned unit numbers beginning with 38, not 12.

Note

The only place that this logical partitioning information is kept is in the Unit Table entries for each volume; however, the information in the Unit Table is used by other parts of the system, such as the MEDIAINIT program that initializes (formats) the disc. The disc drive itself has *no* knowledge whatsoever of this partitioning scheme.

As another example of device priorities, suppose that you had an HP 9133XV drive and an HP 7908 drive in your system. Suppose also that the 9133 is at 702 and the 7908 is at 700. The 9133 is at the higher bus address (and is therefore found after the 7908 is found during the scan sequence), so it has the higher priority (assuming that the 7908 was not the boot device). The standard TABLE presumes that the 9133 is partitioned into 14 logical volumes, so it allocates 14 unit numbers (11-24) for the device. It then allocates 16 unit numbers (25-40) for the 7908 for analogous reasons.

As you might guess from the preceding discussion, even though there may be up to 10 devices in the list of hard discs, not all of the volumes they contain will necessarily be assigned unit numbers and thereby made accessible. Only the volumes to which unit numbers are assigned will be accessible. For instance, if the preceding example would have been two 7908 drives, then the highest priority device will be assigned 16 unit numbers (11-26), while the lower priority drive will only be assigned 14 unit numbers (27-40). If this disc had actually been partitioned into 16 logical volumes, then its last 2 volumes would *not* be accessible.

Note

If you plan to use your hard disc with BASIC, you should set up the disc as one logical volume. See the File Interchange Between Pascal and BASIC section of the Technical Reference appendix.

Choosing the System Volume

The final step made by the standard TABLE is to choose the system volume. The operating system makes use of this volume for several purposes. For instance, after the system volume is designated at boot time, it is then inspected for system files (such as the EDITOR, FILER, and COMPILER). It is where the autostart file (AUTOSTART or AUTOKEYS) is assumed to be. It is also used by the system for storing temporary files that it creates for processes such as expanding Stream files. The system volume should remain on-line at all times if possible.

Here is the algorithm used by the standard TABLE program to determine which volume will be designated as the system volume; the "Boot volume" is the volume from which the BOOT: files named SYSTEM_P, INITLIB, and TABLE were loaded:

1. If a Boot volume was assigned a unit number *and* its capacity is greater than 300 Kbytes, then this device is designated as the system volume.
2. If step 1 did not designate a system volume, then search all volumes in the sysunit_list (a structure declared in TABLE). If a logical volume with a valid directory is found during this search, then it will be designated as the system volume.
3. If neither step 1 or 2 designated a system volume, then use the Boot volume as the system volume.

Failure of the TABLE Program

By the way, if the TABLE auto-configuration program ever fails during the boot process, unit number 6 (normally the standard PRINTER: volume) is assigned to the screen, and unit number 3 is assigned to the "Boot device." You can only execute programs off of unit number 3 (with the Main Level eXecute command); it is otherwise inaccessible to the File System.

If TABLE is executed again but fails during this subsequent execution, then the Unit Table reverts back to its state before this unsuccessful try was attempted. (This is true *every* time that TABLE is subsequently executed.)

Example Special Configurations

This section describes several common types of special configurations. It outlines the general procedures required to implement them. The subsequent section called Modifying the Configuration provides the details of the procedures.

Hard Disc Partitioning

The way that the standard TABLE program prefers to partition hard discs was explained in the preceding discussion of Auto-Configuration. This section briefly describes the two methods of changing this default partitioning.

- "Coalesce" adjacent directories into one larger volume
- Modify the CTABLE program to partition the discs differently

The procedure for coalescing logical hard disc volume is given in the subsequent Modifying the Configuration section.

Coalescing adjacent volumes will work well under these *general* circumstances:

- The total number of volumes that TABLE assumes it will find on *all* discs is less than 30.
- The sizes of volumes that you can make by coalescing an integral number of logical volumes is acceptable (i.e., the "resolution" of the default volume sizes is good enough).

If the desired configuration cannot be made by merely coalescing volumes, then you will have to modify the standard TABLE program (CTABLE.TEXT source file). Modifying the standard TABLE is also described in Modifying the Configuration.

Multiple On-Line Systems

If requested by operator intervention at power-up, computers equipped with Boot ROM 3.0 and later versions find all the on-line system Boot files (for example, SYSTEM_P) and display their names. You can choose the one you want to be booted. For instance, if you have a Pascal and a BASIC system on-line, you can choose which you want to boot.

Note

The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.

The term "BOOT: file" is used to identify a file used during the boot process; these files are on the BOOT: or BOOT2: disc shipped with your system.

By modifying certain BOOT: files (usually INITLIB and TABLE only) and uniquely re-naming each different set, you can give yourself the option of choosing different Pascal system configurations at power-up. (This is not possible with the earlier Boot ROMs.)

For instance, suppose you want to have one system version that sets up SRM as the default volume and another that does not allow access of the SRM system. In such a case, you can create two systems, each of which is tuned for the desired usage; you can choose the one you want at power-up. To configure your system as such, you need to make duplicate copies of some system files and change some of their names. (You also need to set up the SRM system, as described in a later section of this chapter.)

This type of configuration usually requires only the following type of modification to the standard configuration:

- Change file names and copy them to different volumes

You can optionally make this type of configuration change, depending on the hardware available and the desired use of it by the different systems you want to have on-line:

- Add module(s) to INITLIB

This type of change does not usually require changes to the TABLE program.

See the discussion of Copying System Files and Changing Their Names in the Modifying the Configuration section for further details.

Adding Interfaces and Peripherals

Here is a brief summary of how to add several interfaces and peripheral devices to your system.

Hardware Configuration
You should configure each interface according to the instructions given in its installation manual. Most switches can be set to their factory defaults; however, the Pascal documentation will tell you when you will need to change the switch settings from the defaults.

Software Configuration
Using a peripheral device (and the corresponding interface) for File System operations may require this type of change to the standard configuration:

- Add module(s) to INITLIB

You may need to (or optionally want to) perform this type of configuration change:

- Modify the TABLE program

Here is a list of interfaces and peripheral devices and corresponding configuration modifications you will need to make in order to use each one. See the discussion of the type of change that you will make in the Modifying the Configuration section.

- **HP 98620 Direct Memory Access (DMA) Interface**
  The driver for this interface is module DMA, which is present in the original INITLIB. The interface is always used in conjunction with other cards.
- **HP 98622 GPIO (16-bit parallel) Interface**
  To drive this interface with the IO Library, add module GPIO. (GPIO is **not** required if you use the interface only for the HP 9885 disc; however, F9885 is required.) See the *Pascal Procedure Library* manual for further details regarding I/O applications.

- **HP 98624 HP-IB Interface**
  Using this HP-IB interface requires module HPIB, which is already present in supplied INITLIB. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98625 High-speed Disc Interface**
  This is a form of HP-IB interface, but it is only for use with discs and, oddly enough, printers. Modules DMA (already present in the standard INITLIB file) and DISC_INTF (not in the standard INITLIB) are required to use this interface.
- **HP 98626 Serial RS-232 Interface**
  To drive this interface, install module RS232. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98627 Color Output Interface**
  Using this interface is described in the *Pascal Graphics Techniques* manual for further details regarding I/O applications.
- **HP 98628 Data Communication Interface**
  To drive this interface, install module DATA_COMM. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **HP 98629 SRM Interface**
  Using this interface requires that you set-up an SRM system. See the Setting Up the SRM System and the Example SRM Configuration sections of this chapter for further details.
- **HP 98630 Breadboard Card**
  This interface is intended for use only by system designers. See the *Pascal System Internal Document* for details.
- **HP 98635 Floating-Point Math Card**
  No additional modules are required to use this card. However, you will need to use one of the FLOAT_HDW Compiler options; see the Compiler chapter for further details. The FLTLIB:FGRAPHICS module is optimized for use with this card.
- **HP 98644 Serial RS-232 Interface**
  To drive this interface, install module RS232. See the *Pascal Procedure Library* manual for further details regarding I/O applications.
- **Printers**
  Module PRINTER (already present in standard INITLIB) is required to drive all printers ("local" printers, not those on SRM), regardless of the type of interface being used. Additionally, module HPIB (already present in INITLIB) is required for HP-IB printers. Printers with RS-232C interfaces can be used with the HP 98626 RS-232C Serial interfaces if module RS232 is added to INITLIB. To drive a printer with an HP 98628 Datacomm interface, you will need to add module DATA_COMM.

  If a printer with an RS232C interface is to be recognized by the File System (for example, volume PRINTER:), then you will need to modify the TABLE program. See the Changing the System Printer section for further details.

- **Graphics Output and Input Devices**
  To talk to "local" (i.e., non-SRM) HP plotters via HP-IB requires module HPIB (already present in the supplied INITLIB). Modules DATA_COMM and SRM are required if you are using the plotter spoolers on an SRM system. In addition, you will need to use modules in the GRAPHICS library. Normally, you will not access any local plotter through the File System (i.e., you will not access it through a logical unit number); thus, you will not need to add modules to INITLIB or modify the TABLE program. See the *Pascal Graphics Techniques* manual for further details.
- **Mass Storage Devices**
  You will almost always access mass storage devices (such as disc and tape drives, EPROM, and Magnetic Bubble memory) from the File System. The TABLE auto-configuration program finds most "common" disc peripheral devices; however, to use "non-standard" devices like EPROM and Bubble cards, you will need to modify the program. See the corresponding sections of this chapter for further details.

HP-IB Disc Performance Considerations

Disc performance is primarily determined by the device itself, but it may also be affected by the hardware used to interface the disc to the computer. Three common interface usages and their relative performance is given in the table below.

Lowest	Internal HP-IB or HP 98624 HP-IB interface (without a DMA card)
Higher	Internal HP-IB or HP 98624 HP-IB interface (with a DMA card)
Highest	HP 98625 High-Speed Disc interface and an HP 98620 DMA card (the 98625 card *requires* a DMA card)

The 913xA Hard Discs Drives (excluding the V, B, and XV suffix drives) show an increase in performance when a DMA card is used.

Although it is not required, you should use an HP 98625 High-Speed Disc interface with CS80 discs for optimal performance.

While the HP 9121, 9895, 9133 and 9134 discs can be used with the HP 98625 High-Speed Disc interface, they do not realize any increase in performance.

Note

Never use the HP 98625 High-Speed Disc Interface with an HP 82901, 82902, or 9135 disc drive.

Setting Up an SRM System

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like disc drives, printer spoolers, and plotter spoolers. Also, the SRM may be the only mass storage device for a machine with no local disc drives.

This section only briefly explains what is required to configure Pascal workstations in order to access an SRM system. Here are the main steps:

1. Add modules DATA_COMM and SRM to INITLIB and re-boot, or execute them and re-execute TABLE; this step provides minimal access to the SRM through unit #5:
2. Copy files to certain SRM directories, and optionally re-name files; this step allows you to use unit #45: as the system volume and to boot from an SRM (if your computer is equipped with Boot ROM 3.0 or later)
3. Modify the TABLE program, and re-execute it; this step allows you to assign additional unit numbers to the SRM system

Because configuring a Pascal workstation to access an SRM system is not a trivial task, it is used as the example special configuration. See the subsequent section called An Example SRM Configuration (near the end of the chapter) for further details.

Changing the System Printer

Normally, the TABLE program assumes that the "system printer" (the PRINTER: volume, unit #6) is an HP-IB device at select code 7 with primary address 01. This section tells what is required to override this assumption. Here are the general changes you will need to make:

- If the printer is not an HP-IB device, you may need to add the corresponding driver module(s) to INITLIB. See the Adding Modules to INITLIB discussion in the subsequent Modifying the Configuration section.
- Modify the TABLE program so that it sets up the printer as the system printer (volume PRINTER:). See the Local Printer Type Option discussion in Modifying the TABLE Program.

Setting Up Printers with RS-232C Interfaces

The TABLE source program provides a very clean way to set up an RS-232C printer as the PRINTER: volume. Here are the conditions required:

- The RS-232C interface can be an HP 98626 or 98644 RS-232C Serial, a built-in RS-232C Serial, or an HP 98628 Datacomm interface. The default select code is 9, but you can change the `dav` variable (device address vector) to use another select code.
- In order to use the 98626, 98644, or built-in serial interfaces, you will need to add module RS232 to INITLIB
- In order to use the 98628 interface, you will need to add module DATA_COMM to INITLIB.

- The factory default switch settings[1] for these interfaces are as follows:

  Interrupt level set for level 3
  Baud rate set for 2400 baud
  Stop bits switch set for 1 stop bit
  Bits/char. switch set for 8 bits
  Protocol set for XON/XOFF
  Parity set to off

If your printer uses other parameter(s), then set the interface card[2] to match your printer. See the interface's installation manual for switch locations and settings.

The select code of the interface is assumed to be 9; either set the interface to this select code or modify the `sc` parameter in CTABLE to match the select code of your interface.

You may also need to change the `local_printer_timeout` variable to match your printer's characteristics. See the Local Printer Options section in the discussion of the CTABLE program.

Using Bubbles and EPROM

Magnetic bubble memory and EPROM (erasable programmable read-only memory) are both types of non-volatile memory. The Pascal Workstation system allows you to use HP 98259 Magnetic Bubble Memory and HP 98255 EPROM cards as mass storage devices. This section briefly outlines what is required to configure your system to use these Series 200/300 cards. The Non-Disc Mass Storage chapter gives further instructions.

You will normally be accessing these cards as mass storage devices. Here are the general steps required to make these devices accessible to the File System:

- Add the appropriate driver module(s) to INITLIB:

  To use a Bubble card, add the BUBBLE module to INITLIB. This module adds both read and write capabilities for Bubble cards to the system.

  To read EPROM cards (which have already been written), add the EPROMS module to INITLIB. (To program EPROMs requires an extra step, described below.)

- Modify the TABLE program so that it assigns a logical unit number to the device(s). See the discussion of Table Entry Assignment Templates in the Modifying the TABLE Program section.

See the the Non-Disc Mass Storage chapter for the complete description of using these cards.

1 The 98644 card has no baud rate or line control switches, so the Pascal system sets these default paramters.

2 With the 98644 card, you may need to write a short application program to set the parameters. See the *Pascal Procedure Library* manual for details.

Using Alternate DAMs

The files on a disc are found and accessed by means of a directory which describes where the files are located, how big they are, what types of data they contain, etc. The directory is stored on the disc itself. There are many reasonable ways to organize discs, depending on one's purposes. The methods of accessing these alternative organizations are called "Directory Access Methods", or DAMs. A mass storage volume can be read or written by the File System only if the correct DAM is used.

Pascal 2.0 and later versions support three mass storage directory organizations: the Workstation Pascal 1.0 format (WS1.0, similar to UCSD format); HP's Logical Interchange Format (LIF); and the Shared Resource Manager's (SRM) hierarchical, or "structured," directory format (SDF).

In the case of Shared Resource Management discs, the DAM is supported in the SRM itself; what Pascal supports is the communication of DAM requests to the SRM. The SRM method can only be used with remote mass storage over an SRM hookup. The other two methods can be used with any local mass storage device.

The DAM used for each logical unit is selected by the TABLE configuration program. The standard TABLE selects LIF as the "primary" DAM, and UCSD (Workstation 1.0 compatible) as "secondary". (Sometimes the word "alternate" is used instead of "secondary".) The primary DAM is the one used for *blocked* units in the range of #1 through #40 (except #5, auto-configured as the SRM unit). The secondary DAM is used by blocked units in the range of #43 through #50 (with these exceptions: #45 is auto-configured as the SRM system unit, if appropriate; memory volumes always use the primary DAM).

This secondary DAM is available to allow discs with the secondary directory format to be used by Pascal programs and the Filer utility. The secondary DAM is in no way restricted from normal use by the File System; discs in the secondary DAM units can be read and written directly by Pascal programs.

If Pascal 3.0 is booted up just as shipped, the primary DAM will be LIF. In this case, Pascal 1.0 discs must be accessed through the alternate DAM units. To make the Pascal 1.0 format the primary DAM, you will need to change the TABLE program. See the section called Modifying the Configuration Table later in this chapter for further details.

CAUTION

DO NOT MAINTAIN BOTH LIF AND UCSD DIRECTORIES ON ONE LOGICAL VOLUME. THE DIRECTORIES HAVE NO KNOWLEDGE ABOUT THE OTHER'S EXISTENCE, SO EACH CAN READILY DESTROY DATA IN THE OTHER DIRECTORY.

Comparison of LIF and WS1.0 DAMs.
In both DAMs, all the space allocated to a single file is contiguous. Consequently, if the free disc space is fragmented, either DAM may be unable to create a new file of a specified size even though there is enough total free space on the disc.

In either DAM, it may not be possible to extend (append to) an existing file even if the disc volume has some free space. A file in either DAM can only be extended if there happens to be free space immediately following the file. Appending to files was not allowed in Pascal 1.0.

Letter case is significant in LIF file identifiers; for instance, the file called "Charlie" is not the same as "charlie". Letter case is not significant under the Workstation DAM (more precisely, file names are automatically converted to upper case in Workstation disc directories). The same comments apply to volume names in the two DAMs.

Workstation DAM file names may be up to 15 characters long. LIF names are restricted to 10 characters. In many cases this difference need not be a problem. Most file names used by the Pascal system end in a five-character suffix such as ".TEXT" and ".CODE"; hence the useful part of such names is 10 or fewer characters. The LIF DAM implementation encodes recognized standard suffixes into the suffix of a LIF file name, so that nine characters are available for the significant part of the name. This encoding is transparent, as is the decoding back into the full suffix when necessary.

Recommendations For Selecting Primary DAM.
If you are a new user and have no existing discs in the WS1.0 format, *we recommend that you use the system as supplied, with LIF as the primary DAM.* LIF is an HP standard for information interchange among computer systems. For instance, the BASIC and HPL systems that run on your Series 200 computer use LIF directories. The boot device must have a LIF directory unless you are booting from SRM.

If Pascal 2.0 or later system version is booted as shipped, the primary DAM will be LIF. If you have discs generated by Pascal 1.0 you have three choices.

- Change the primary DAM to the Pascal 1.0 directory.
- Adopt LIF for new volumes but access your Pascal 1.0 directories through the limited number of alternate DAM units.
- Transfer your old files on Pascal 1.0 volumes to new LIF volumes.

The choice is primarily one of convenience, although in the long run there may be some advantages to LIF. Since Pascal programs which ran under the 1.0 release must be recompiled to run under Pascal 3.0, you may choose to convert your discs as well.

Moving Files Between WS1.0 and LIF Volumes.
The following steps outline the method of moving files from one directory type to another.

1. Put the ACCESS: disc in a disc drive and press [F] to run the Filer.
2. Put the source disc in a drive configured for its type of DAM, and the destination disc in a drive configured for its DAM. (See below)
3. Use the Filer's Filecopy command to move files from one disc to the other. The Filer commands will work with either DAM.

Note that the alternate DAM units allow either DAM to be used in the same drive. For instance, you can copy a file from #43 to #3, both of which are assigned to the right-hand flexible disc drive in a Model 236 or 226 computer. For example:

Press [F] (for Filecopy), then enter:

```
#43:CHARLIE,TEXT,#3:$
```

The Filer will tell you to when to swap discs.

Remember that the name of a file in a WS1.0 directory may be too long for a LIF directory. You may have to invent a shorter name.

By the way, it's a good idea to develop the habit of using uppercase letters in the names of LIF files, because some other systems will not allow or recognize file names with lowercase letters.

Note that directories created by Pascal 1.0 have either 77 or 233 entries, whereas WS1.0 directories created by Pascal 2.0 and later versions have a variable number of entries specified by the user. Thus you can use Pascal 2.0 and later versions to create WS1.0 format directories which aren't readable by Pascal 1.0; whereas all Pascal 1.0 directories are readable by Pascal 2.0 and later versions.

Notes

Modifying the Configuration

This section describes the mechanics of modifying the "standard" configuration methods of the system as it was shipped to you. Here are the general methods of modifying the standard configuration:

- Coalescing adjacent hard disc volumes
- Copying system files and changing their names
- Using AUTOSTART and AUTOKEYS Stream files
- Adding driver modules to INITLIB
- Modifying the standard TABLE program

Coalescing Hard Disc Volumes

As discussed previously, you can manually coalesce adjacent logical volumes on hard discs. For instance, suppose that you have an HP 9133V hard disc drive which is partitioned into the standard 4 logical volumes; the standard volume size is approximately 1 Megabyte. However, you want to increase the size of the first logical volume. You can easily coalesce the second volume with the first to double the size of the first. (This type of change is *only* possible with Option 10 machines; machines without this option cannot be logically partitioned.)

Overview of the Example Procedure

To coalesce the two logical volumes in this example, here are the steps you will take.

1. If the disc has not been initialized, then you will need to do so before continuing with this procedure.
2. Invalidate the directory of the second volume by overwriting it with the data in a file. (Since the purpose of this step is to invalidate the directory, this file must *not* resemble a directory.)
3. Change the Unit Table by running the standard TABLE program. TABLE will find the invalid second directory, invalidate the corresponding Unit Table entry, and enlarge the volume size parameter of the preceding Unit Table entry (the first volume's Unit Table entry). This step sets up the Unit Table in preparation for coalescing the two volumes. Note that the first volume's directory *on the disc* has *not* been changed at this point; it is still the original size.
4. Create a new directory on the disc for the first volume; this directory will reflect its new size. To do this, you will need to destroy the first directory on the disc and then use the Filer's Zero command to zero it. The Zero command will read the size for the first volume *from the Unit Table*, since it will not have found a valid directory on the disc. The two volumes will then be "coalesced" *on the disc* when the first directory is enlarged as it is zeroed.

Prerequisites

You should perform this operation *before* placing any valuable data in the volumes to be coalesced; however, if you have already used the volume, then you can back-up these files on another volume (such as a floppy disc or another hard disc drive). Once a volume has been coalesced with another, any data in it *cannot* be accessed.

The Example

The following procedure is an example of coalescing the second volume of an HP 9133V hard disc with the first volume, which results in approximately a 2-Megabyte first volume; the original third and fourth volumes will be left at the default size of approximately 1 Megabyte.

1. If the disc to be partitioned (here, the 9133V) has not already been installed with switches set properly, do so now. Set the drive's HP-IB address to a value that will ensure that it has a high enough priority to be assigned unit numbers. (Device Priority is fully discussed in The Booting Process at the beginning of this chapter.) For this example, we will assume that it will be assigned unit numbers 11 through 14.
2. If the disc has not already been assigned unit numbers (such as during a previous boot sequence), then use the eXecute command to run the standard TABLE program. If this program is not currently on an on-line volume or P-loaded into memory, then you will need to insert the BOOT: disc into a drive. Press [X] at the Main Command Level. The system prompts with this question:

   ```
   Execute what file?
   ```

   Enter the file specification of TABLE; the following specification indicates that it is on the BOOT: volume:

   ```
   BOOT:TABLE.
   ```

   The trailing period is required to suppress the otherwise automatic ".CODE" suffix, since this file's name on the disc has no suffix.

   When TABLE has finished, the disc should have been found and assigned unit numbers (we will assume 11 through 14). However, if it has not been previously initialized and directories zeroed, then unit numbers assigned to it will not show up in a Filer's Volumes command (they are invalid because the corresponding directories were found to be invalid).
3. At this point there are two main situations possible: the disc either has or has not been initialized.
   a. If it has *not* been initialized, do so now; proceed with step 4.
   b. If it has already been initialized, you have two more alternatives.

      If volumes have been coalesced and you want to split them (or if it is a disc initialized as one large volume), then you will need to first partition it into smaller volumes. Proceed with step 5.

      If it is has already been initialized but is still partitioned into the default number of volumes (with default sizes), or if it has volumes which have been coalesced and you don't need to split them, then you can proceed to step 6.

4. If the disc has *not* yet been initialized, then you will need to initialize it now.

   Since the disc has not been initialized, the standard TABLE program will not have found *any* valid directories at expected locations on the disc, and therefore will assume that it is to be partitioned into the default number of volumes (shown in the discussion of partitioning given in The Booting Process early in this chapter). It will build the Unit Table accordingly.

   a. Put the ACCESS:MEDIAINIT.CODE program on-line, and press [X] to execute it. The system responds:

      ```
      Execute what file?
      ```

      If the program is on the ACCESS: disc, enter:

      ```
      ACCESS:MEDIAINIT
      ```

      The ".CODE" suffix will automatically be appended to the file name.

      The program prompts for a unit number:

      ```
      Volume ID?
      ```

      Enter the unit number of the first volume on the disc that is to be initialized. In this case, enter:

      ```
      #11:
      ```

      The program asks for verification:

      ```
      Device: 913xA series hard disc, 707, 0
      Logical unit #11 - < no directory >

      WARNING: the initialization will also destroy:
        #12: < no dir >
        #13: < no dir >
        #14: < no dir >

      Are you SURE you want to proceed? (Y/N)
      ```

      The 913xA corresponds to the 913x "V" suffix drives. The select code and HP-IB address (707), and drive number (0) should also correspond to the disc to be initialized. If not, then answer [N] and correct the problem. If this is the disc you want to initialize, then answer affirmatively by pressing [Y]. The program then displays:

      ```
      Medium initialization in progress
      ```

      The program takes about 15 minutes to initialize this type of disc.

      After the program has finished, it displays this message:

      ```
      Medium initialization completed
      ```

      Each volume's directory is then "zeroed" (cleared, named, and validated):

      ```
      Volume zeroing in progress
      ```

      Here is the message that indicates the entire initialization and zeroing is successful:

      ```
      Volume zeroing completed
      ```

b. Verify that the volumes are have been zeroed and are accessible by using the Filer's Volumes command. Press [V], and the Filer shows you the volumes currently on-line:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 * V11:
 12 * V12:
 13 * V13:
 14 * V14:
Prefix is - BOOT:
```

b. If all volumes have been initialized and directories zeroed properly, proceed to step 6.

5. If volumes on the disc have been coalesced and you want to split them (or if the disc was initialized as one large volume), then you will need to restore part or all of the default partitioning structure now.

   To do this, you will need to destroy the existing directories on the disc which are to be split. For instance, if your disc is one large logical volume, then you merely need to destroy the first directory, since it is the only directory that currently exists on the disc.

   a. To destroy a directory on the disc, you will overwrite the directory with a file (the MEDIAINIT.CODE file will work for this purpose). Make sure that the Filer is on-line and then invoke it by pressing [F] from the Main Level. Then use the Filecopy command to overwrite the directory with the file; press [F] and the following prompt is displayed:

      ```
      Filecopy what file?
      ```

      Enter:

      ```
      ACCESS:MEDIAINIT.CODE
      ```

      It then asks:

      ```
      Filecopy to what?
      ```

      You will answer:

      ```
      #11:
      ```

      The Filer verifies with this prompt:

      ```
      Destroy directory V11 ?
      ```

      Affirm that you want to destroy the directory by pressing [Y].

      The Filer then shows that it has made the requested copy:

      ```
      ACCESS:MEDIAINIT.CODE        ==> #11:
      ```

b. If other existing directories on the disc are to be split, then destroy each by repeating this process.

c. After destroying all directories to be split, run TABLE again to restore the Unit Table to set up the default partitioning. This step does **not** partition the disc; it "partitions" the Unit Table in anticipation of the subsequent disc partitioning.

d. Now partition the disc by zeroing volumes #11, #13, and #14. If you are not still in the Filer, put it on-line and press [F]. Use the Zero command to zero the first volume on the disc. Press [Z]. The Filer responds with this prompt:

```
Zero directory (NOT valid for SRM type units)
Zero what volume?
```

Enter the unit number of the first volume:

```
#11:
```

The Filer then responds:

```
Destroy V11 ? (Y/N)
```

Press [Y] to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

```
Number of directory entries?
```

If you want a number other than the default (80), then enter it now; otherwise, press [Return] or [ENTER] to accept the default.

The Filer next prompts for the volume size. The number shown in parentheses is the default:

```
Number of bytes (1206272) ?
```

Accept the default by pressing [Return] or [ENTER].

Finally, you will be prompted for the new volume name:

```
New directory name ?
```

Enter any valid volume name of up to 6 characters. For this example, enter:

```
V11:
```

The Filer verifies that it has the name you requested:

```
V11: correct ? (Y/N)
```

Press [Y] to confirm the name, if it is correct. If is not, then answer [N]; you will need to start the Zero command again.

If you confirmed the name, the Filer shows that the directory was created:

```
Volume V11 zeroed
```

e. Repeat this process for each unit number to Zero all directories which you want to *remain* on the disc; you need not Zero those that will be coalesced later in this procedure.

6. If the second directory exists (unit #12), then destroy it. If it does not exist, then proceed to step 7.

   To destroy a directory, use the Filer's Filecopy command to copy a file into the directory (the MEDIAINIT.CODE file will work just fine for this purpose). Make sure the Filer is on-line, and press [F] to enter the Filer. Invoke the Filecopy command by pressing [F]. It prompts:

   ```
   Filecopy what file?
   ```

   Enter the specification of the MEDIAINIT file:

   ```
   ACCESS:MEDIAINIT.CODE
   ```

   The system promts:

   ```
   Filecopy to what?
   ```

   Enter the specification of the second directory:

   ```
   #12:
   ```

   Since a directory already exists on this volume, the Filer prompts to see if you really want to proceed and destroy this directory:

   ```
   Destroy directory V12 ? (Y/N)
   ```

   Type [Y] to enter an affirmative response. The Filer then shows that it completed the operation by displaying this message:

   ```
   ACCESS:MEDIAINIT.CODE              ==> 12:
   ```

7. Now you should execute TABLE again. This execution of the program will find no second directory, and consequently will make the Unit Table entry for the first directory reflect the size of both first and second volumes (about 2 Megabytes). No changes to the disc will be made by this step, however.
8. Now destroy the first directory and then Zero the volume. Destroying this directory is necessary in order for the Zero command to read the size of the volume *from the Unit Table.* If it is *not* destroyed, then the volume size will be read from the disc and the volumes will *not* be coalesced; the first directory will retain its original size.

   a. Use the Filecopy command to overwrite the first directory. While in the Filer, press [F]. The Filer prompts:

      ```
      Filecopy what file?
      ```

      Answer:

      ```
      ACCESS:MEDIAINIT.CODE , #11:
      ```

      The Filer then asks:

      ```
      Destroy directory V11 ? (Y/N)
      ```

      Answer [Y] to affirm that you do want to destroy it.

b. Finally, Zero unit #11's directory. Press [Z], and the Filer prompts:

```
Zero directory (NOT valid for SRM type units)
Zero what volume?
```

Answer:

```
#11:
```

The Filer then responds:

```
Destroy V11 ? (Y/N)
```

Press [Y] to confirm the command. The Filer then prompts for the number of file entries to be contained in the directory:

```
Number of directory entries?
```

If you want a number other than the default (80), then enter it now; otherwise, press [Return] or [ENTER] to accept the default.

The Filer next prompts for the volume size. The number shown in parentheses is the default (note that it is now twice its former size):

```
Number of bytes (2412544) ?
```

Accept the default by pressing [Return] or [ENTER].

Finally, you will be prompted for the new volume name:

```
New directory name ?
```

Enter any valid volume name of up to 6 characters. For this example, enter:

```
V11:
```

The Filer verifies that it has the name you requested:

```
V11: correct ? (Y/N)
```

Press [Y] to confirm the name, if it is correct. If is not, then answer [N]; you will need to start the Zero command sequence again.

If you confirmed the name, the Filer shows that the directory was zeroed:

```
Volume V11 zeroed
```

9. After zeroing has completed, verify that the disc is as partitioned as desired.
   a. Use the Filer's Volumes command to verify that there are only volumes #11, #13:, and #14: on the disc.

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 * BOOT:
  6   PRINTER:
 11 # V11:
 13 # V13:
 14 # V14:
Prefix is - BOOT:
```

   If you don't have the partitioning scheme that you want, you may have a mistake during the procedure. You may need to repeat the appropriate part(s) of the procedure.

   b. Use the Filer's List_directory command to verify that volume #11: is now larger. Look for the number of available sectors (it should be approximately as shown below).

```
V11:                Directory type= LIF level 1
created 15-May-84   1,31,24 block size=256
 Storage order
...file name....     # blks     # bytes   last chng

FILES shown=0 allocated=0 unallocated=80
BLOCKS (256 bytes) used=0 unused=9412 largest space=9412
```

   Note that the number of entries you specified for the directory will affect the number of sectors usable for files. This example shows the number of sectors left after allocating space for 80 directory entries (files). You will have fewer sectors usable for files if you specified a greater number of entries.

   If the size is not what you expected, then you may have made a mistake during the procedure. If so, you will need to repeat the appropriate part(s) of the procedure.

Copying System Files and Changing Their Names

One of the easiest ways to change the configuration of your system is to copy files from the flexible discs on which it is shipped to discs with better performance (such as local hard discs or SRM discs). This section describes several things to consider while making this type of modification to your system.

Copying Files to the System Volume

If you have a system with one hard disc (such as a CS80 or 913x hard disc), an SS80 flexible disc (such as the 9122), or an eight-inch flexible disc (such as the 9895), then that device may be selected as the system volume during the boot process. (The system volume and its uses are described in the *Pascal User's Guide* and in the File System and Filer chapters of this manual.) It is often useful to copy the most-used of the following "system files" to this system volume to increase performance.

```
EDITOR
FILER
COMPILER
LIBRARY
LIBRARIAN
ASSEMBLER
```

If you P-load these files, then you may not want them to take up room on your system volume; you may want to put them on another less-used volume.

As mentioned at the beginning of this chapter, the following files are used during the boot process. They are on the standard BOOT: disc shipped with your system.

```
SYSTEM_P
INITLIB
TABLE
```

If you have Boot ROM 3.0 or later versions (and not 3.0L), these BOOT: files may be placed together on any volume you choose, provided it has a LIF or Shared Resource Management (SRM) directory. They can also be renamed; the purposes of and conventions for renaming these BOOT: files will be described later in this section. If you do not have Boot ROM 3.0 or later version (which is only possible with earlier 9826 and 9836 computers), these files must all be on the right-hand internal disc drive. (The method of determining which Boot ROM you have is described in the *Pascal User's Guide.*)

The STARTUP file is also used during the boot process, but it can either be on the boot volume or on the system volume. You might leave it on the boot volume if there isn't room on the system volume. However, if possible, put it on the system volume so that it loads faster.

If you have an HP 9885 8-inch disc drive as the system device, not all the system files will fit on it. Using the above procedure, copy onto it those system files which are used most frequently (such as EDITOR, FILER, and COMPILER).

Default BOOT: File Names

On the BOOT: discs shipped from the factory, the files used during the boot process are named as follows:

```
SYSTEM_P
TABLE
INITLIB
STARTUP
```

If these files are copied to another local (non-SRM) mass storage volume and the names are retained, they will boot normally.

Re-Naming the BOOT: Files

If you change the name of SYSTEM_P (the system Boot file), then you must also change the names of some other files on the BOOT: disc. The advantage is that the same Boot file (with different names) can load specialized BOOT: files for unique hardware configurations.

Note

The term "BOOT: file" is used to identify a file used during the boot process; these files are on the standard BOOT: or BOOT2: disc shipped with your system.

The term "system Boot file" is used to identify a file that is found and loaded by the Boot ROM, such as SYSTEM_P; this file then loads the corresponding operating system.

The Boot ROM 3.0 and later versions also recognize system Boot files if the file name begins with "SYS". If you rename the Pascal system Boot file (SYSTEM_P), there are file naming rules you must follow so the system Boot file can identify the other BOOT: files. The rules for BOOT: file names are as follows:

- If the complete string "SYSTEM_" is used in the system Boot file name, up to the next three letters of the file name are added to the base of the other BOOT: file names (INIT, TABLE, START).
- If only "SYS" is used in the system Boot file name, up to the next seven letters of the file name are added to the base of the other boot file names (possible only with Boot ROM 3.0 and later versions).

For example, if you change the system Boot file's name from SYSTEM_P to SYSTEM_P3 (for Pascal 3.0), then the Boot file will look for the following files:

```
INITP3
TABLEP3
STARTP3
```

Keep in mind that file names on a LIF directory must be 10 characters or less.

If you change the name to Boot file's name SYS_SRM_P3, it will look for the following files:

```
INIT_SRM_P3
TABLE_SRM_P3
START_SRM_P3
```

File names on SRM directories can be up to 16 characters.

In general, SYSTEM_P could be the Pascal Boot file that loads the standard Pascal BOOT: files. SYSTEM_P3 is the same Boot file, but INITP3 and TABLEP3 could support the hard discs. SYS_SRM_P3 is the same Boot file, but INIT_SRM_P3 and TABLE_SRM_P3 could support SRM.

Normally, a special TABLE is not required for hard discs or SRM systems, although you may wish to create one for a special application.

SRM users needing a special TABLE should use the "private" system volume based on the node address mechanism to keep their own custom TABLE and INITLIB (i.e., the default system volume when booting a workstation from SRM is the /WORKSTATIONS/SYSTEM*nn* directory, in which *nn* is the node address of the workstation). See the subsequent Example SRM Configuration section for further details.

AUTOSTART and AUTOKEYS Stream Files

As discussed in a previous section, Stream files allow execution of commands just as if you had entered them from the keyboard. When you put a Stream file named AUTOSTART on your system volume, the keyboard commands the file contains are automatically executed during the booting process; if the volume is a read-only device, such as EPROM, then you should call the stream file AUTOKEYS. (The Stream command is fully described in the "Main Command Level" chapter.)

You can use autostart files to perform such functions as the following: load drivers; use the What command to change the system files, system library, default or system volume; re-execute the TABLE program (or execute another like it). Be aware that there are also other ways to perform this type of configuration; however, this method can be used to quickly or temporarily change the configuration by creating different stream files, renaming the one you want to be used as the autostart file to AUTOSTART (or AUTOKEYS), and then re-booting.

In order to configure your system to access an HP 98626 RS-232C Serial interface, for instance, you can install the driver module with an autostart file. Here is an example stream file that performs this installation (this example assumes that the file named RS232 is on the volume that is chosen as the system volume at power-up):

```
<blank line>
<blank line>
x*RS232.
v
```

The two blank lines which occur first are two carriage returns in the file which are "null" responses to the Time and Date prompts at power-up. These null responses get you to the Main Command Level.

The "x" in the first column of the third line is an eXecute command. The period at the end of the file name prevents the system from appending ".CODE" to the file name. The "v" at the end of the file is the Version command. It gives you another chance to type in the time and date.

After you've created your AUTOSTART file, be sure that you store it on the system volume. This is done by Quitting the Editor, selecting the Write option, and entering this file specification:

```
*AUTOSTART.
```

The period at the end of the file name prevents ".TEXT" from being added to the file name. If you were a Pascal 1.0 user, the file was called AUTOSTART.TEXT. With Pascal 2.0 and later versions, it is called AUTOSTART. Notice that uppercase characters must be used.

Adding Modules to INITLIB

As mentioned previously, the INITLIB supplied on your original BOOT: or BOOT2: disc contains a reasonably complete set of peripheral driver software. You may wish to install other drivers, which are supplied on the CONFIG: disc; or to conserve memory you may wish to remove items you don't need.

Unlike the System Library, modules in INITLIB are order sensitive. Certain modules, if present, must precede others in INITLIB. The list which follows shows the recommended order of all the "driver" modules supplied with Pascal 3.1. If you add or delete INITLIB modules, all the modules which are present in the resulting INITLIB should appear in the order listed.

Required Order of Modules in INITLIB

The table lists the importance of each module. Items marked "Required" are essentially required in INITLIB. Items marked "Almost" are almost always required. These modules should not be removed unless you have determined for sure they aren't needed, because they are part of the normal functioning of the system. Items marked "Development" are usually needed in a software development environment. Items marked "Optional" are optional unless required by a particular system configuration.

Required Order of Modules

Module	Where found	Importance
KERNEL	BOOT:INITLIB	Required
SYSDEVS	BOOT:INITLIB	Required
CRT	BOOT:INITLIB	Required
CRTB	BOOT:INITLIB	Required
CRTC	BOOT2:INITLIB	Required
CRTD	BOOT2:INITLIB	Required
A804XDVR	BOOT:INITLIB	Required
KEYS	BOOT:INITLIB	Required
NONUSKBD1	BOOT:INITLIB	Required
NONUSKBD2	BOOT:INITLIB	Required
BAT	BOOT:INITLIB	Required
CLOCK	BOOT:INITLIB	Required
PRINTER	BOOT:INITLIB	Development
DISCHPIB	BOOT:INITLIB	Development
AMIGO	BOOT:INITLIB	Optional
CS80	BOOT:INITLIB	Optional
IODECLARATIONS	BOOT:INITLIB	Required
HPIB	BOOT:INITLIB	Almost
DMA	BOOT:INITLIB	Development
REALS	BOOT:INITLIB	Required
ASC_AM	BOOT:INITLIB	Development
WS1.0_DAM	BOOT:INITLIB	Development
TEXT_AM	BOOT:INITLIB	Almost
CONVERT_TEXT	BOOT:INITLIB	Almost
LIF_DAM	BOOT:INITLIB	Almost
CHOOK	BOOT:INITLIB	Optional
DEBUGGER	ASM:DEBUGGER	Development
DISC_INTF	CONFIG:DISC_INTF	Optional
DATA_COMM	CONFIG:DATA_COMM	Optional
GPIO	CONFIG:GPIO	Optional
RS232	CONFIG:RS232	Optional
SRM	CONFIG:SRM	Optional
F9885	CONFIG:F9885	Optional
BUBBLE	CONFIG:BUBBLE	Optional
EPROMS	CONFIG:EPROMS	Optional
EDRIVER	CONFIG:EDRIVER	Optional
SEGMENTER	CONFIG:SEGMENTER	Optional
HPHIL	CONFIG:HPHIL	Optional
MOUSE	CONFIG:MOUSE	Optional
DGL_ABS	CONFIG:DGL_ABS	Optional
LAST	BOOT:INITLIB	Required

Note: The BOOT2:INITLIB file contains most of the modules in the BOOT:INITLIB files (the only exceptions are the CRT, CRTB, CHOOK, and BAT modules).

Individual Module Descriptions

Here are brief descriptions of each of the above modules.

- KERNEL is the "core" of the system, containing the Library facility for the Linking Loader and basic File System support. It is always required.
- SYSDEVS, CRT, CRTB, A804XDVR, KEYS, NONUSKBD1, NONUSKBD2, BAT, and CLOCK are responsible for the CRT display, keyboard, foreign character set, battery backup, and clock. They are broken out into several small modules so they may be replaced individually if desired.

  Module CRT is only needed if your computer has an alphanumeric display that is separate from the graphics display (this is the case for most Series 200 computers). Module CRTB is only required in computers with Series 200 bit-mapped graphics displays (such as the HP 9837). Module CRTC is required in all Series 300 computers. Module CRTD is also required in Series 300 computers when the 98700 Display Controller is used.

  Modules NONUSKBD1 and NONUSKBD2 need only be present or replaced by code with equivalent function if foreign keyboards are used.

  Module CLOCK is only required for *programmability* of the battery back-up hardware (this hardware is optional in Series 200 computers; a battery-backed, real-time clock is standard in Series 300 computers).

Note

You can also IMPORT the SYSDEVS module (i.e., use data objects and code declared in it), which is described in the Procedure Library manual. This is the only system module that is described fully enough in this manual set to use in this fashion.

- PRINTER is required to drive all printers, regardless of the type of interface electronics being used. It supports serial as well as HP-IB printers; however, you will have to use the RS232 driver module in order to use printers with RS-232C interfaces. You may also have to modify the variable named `local_printer_timeout` in the CTABLE program; see the discussion of modifying CTABLE later in this chapter.
- DISCHPIB, AMIGO, and CS80 modules are related. To use any external disc drive connected via HP-IB you must use the following modules:

  DISCHPIB and AMIGO for these disc models:

  9895 (8-inch flexible disc)
  9121 (single-sided 3.5-inch flexible disc)
  913x (small hard discs)
  8290x (5.25-inch flexible disc)

  DISCHPIB and CS80 for fixed discs of the Command-Set/80 (CS80) and Sub-Set/80 (SS80) disc drives, and DC600 tape drives, including these models:

  79xx (large hard discs; optional integrated DC600 tapes)
  9122 (double-sided 3.5-inch flexible disc)
  9144 (stand-alone DC600 tape drive)

- IODECLARATIONS is the lowest level of device IO support. Although it is possible to construct loadable systems without this module, only the internal disc drives on the Model 226 and Model 236 can be accessed.
- HPIB is the lowest level support for the Hewlett-Packard Interface Bus, which is HP's implementation of the IEEE-488 Standard. HP-IB interfaces include the built-in HP-IB and HP 98624 cards. Most HP peripherals have HP-IB interfaces, so you will rarely remove this module.
- DMA is the module which runs the HP 98620 Direct Memory Access interface card. DMA provides very high speed data transfers. It is also required in order to use the HP 98625 Disc Interface.
- REALS is the floating-point mathematics support package. It also supports the HP 98635A Floating-Point Math card.
- ASC_AM is the Access Method responsible for blocking and unblocking text files with the LIF-ASCII structure (.ASC files). LIF stands for Logical Interchange Format, a common file interchange structure supported by many HP computers. Since this is one of the formats used by the BASIC language system, it is a good thing to have around. It is also the format used by the SRM for spooled printer files.
- WS1.0_DAM is the Directory Access Method used by the Pascal 1.0 system, a predecessor to the one you are using. This module lets the system read and write discs in that format. Note that the WS1.0 disc organization is compatible with discs written by UCSD Pascal systems; but to read discs written by non-HP computers, a special disc driver is usually required. This DAM can be removed if you have no need to read or write discs compatible with the Pascal 1.0 system.
- TEXT_AM is the Access Method used to block and unblock text files created with the ".TEXT" suffix. These are the files normally created by the Editor (unless the user specifies otherwise). The ".TEXT" file structure is compatible with text files generated by UCSD Pascal systems.
- CONVERT_TEXT is a module used by the Compiler and other subsystems to convert among the various representations of text files. It should be present in INITLIB.
- LIF_DAM is the Directory Access Method required to read and write HP Logical Interchange Format disc directories. LIF is the primary directory organization used with Pascal 2.0 and later system versions, so this module is normally present. If you configure your system to use WS1.0 as the primary directory method (as described in the Special Configurations chapter), you may remove LIF_DAM.
- CHOOK is the display driver for the HP 9836C.
- DEBUGGER is the interactive debugging tool. It is not part of INITLIB (as in pre-3.0 system versions) due to disc space and because it is a particularly dangerous thing to put in the hands of non-programmers. Module REVASM is also a handy tool to have while debugging programs; it allows you to display the contents of memory locations as Assembler language instructions (i.e., "reverse assemble" them).
- DISC_INTF and DMA modules are required in order to use the HP 98625 High-Speed Disc interface.
- DATA_COMM is the module required to drive HP 98628 Data Comm and HP 98629 SRM interfaces.
- GPIO is the module required to drive the HP 98622 GPIO (16-bit parallel) interface.

- RS232 is the module required to drive the HP 98626 and 98644 RS-232C Serial interfaces, and the built-in serial interface in the Model 216 (HP 9816) computer.
- SRM is required to drive the HP 98629 SRM interface. Module DATA_COMM is also required when using SRM.
- F9885 is required for model 9885 flexible disc drives. These discs also require the DMA module, HP 98620 DMA card, and HP 98622 GPIO interface.
- BUBBLE is the module that drives HP 98259 Magnetic Bubble Memory cards. In order to use these cards for mass storage, you will need to add this module to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- EPROMS is required to access the HP 98255 EPROM cards. In order to use these cards for mass storage, you will need to add this module to INITLIB, modify the TABLE program, and program the EPROMs using the HP 98253 EPROM Programmer card (which requires the EDRIVER module). See the Using Bubbles and EPROMs section for further details.
- EDRIVER is the module required to program EPROMs using the HP 98253 EPROM Programmer card. In order to use EPROM cards for mass storage, you will need to add this module to INITLIB and then modify the TABLE program. See the Using Bubbles and EPROMs section for further details.
- SEGMENTER provides the ability to segment programs and run each separately. See the Segmentation Procedures chapter of the *Pascal Procedure Library* manual for further details.
- HPHIL provides the drivers for HP Human Interface Link devices; it is an extension to the A804XDVR module. You can remove it if your computer does have not one of these devices (for example, a 46020 keyboard or a mouse input device).
- MOUSE provides a driver for the optional "mouse" input device, which can be connected to the computer through the HP Human Interface Link (HP-HIL). The driver supports using the mouse for cursor-movement input in both horizontal and vertical directions; it also defines the buttons on the mouse as (Return) or (ENTER) and (Select) ((EXECUTE)). You can access the mouse from your own applications programs; see the Interactive Graphics chapter of the *Pascal Graphics Techniques* manual for details.
- DGL_ABS provides DGL support for the 46087 and 46088 graphics tablets and 35723 TouchScreen input devices.
- LAST is required in every case, and *must* be the last module in INITLIB. The purpose of this module is to actually start the system running after the contents of INITLAB have been loaded and installed in memory. LAST principally does two things: it loads and executes the configuration file called TABLE; and it loads and executes a file called STARTUP, which is usually the Command Interpreter but may be a user program.

A Note About CONFIG:INTERFACE File

INTERFACE contains only the interface text of operating system modules in INITLIB (the code was loaded at boot time). You will need to make this interface text available to the Compiler when you import any of the modules in INITLIB; you can do this by copying the module to be imported from CONFIG:INTERFACE to the System Library or using a SEARCH Compiler option that specifies the INTERFACE file. A good example is importing the SYSGLOBALS module, which requires that INTERFACE be accessed by the Compiler. See the *Pascal Procedure Library* manual for further details.

Note

INTERFACE also contains the interface text of the SYSDEVS module. Using procedures, etc. from this module is described in the *Pascal Procedure Library* manual. Access to most other modules in INTERFACE is not described in the current Pascal documentation set.

Steps for Adding Modules to INITLIB

For this example, we will add module RS232 (on the CONFIG: disc) to the INITLIB file that you are now using to boot your system (possibly on the BOOT: disc); actually, you will create a new INITLIB that includes all existing drivers plus these additional driver modules. These are the modules required to access the HP 98626 and 98644 RS-232C Serial interface cards.

Here is an outline of the procedure that you can use for adding driver modules to the INITLIB library file. It is a straight-forward usage of the Librarian.

1. Set up mass storage. You will need enough on-line mass storage to store *two* copies of the INITLIB file: one for the source (existing) copy, and one for the destination (new) copy. This requirement is made because the new copy of the INITLIB file must not be taken off-line during the whole process.

   To satisfy this requirement, you will need one of the following configurations: one disc large enough to store both (such as a hard disc or SS80 flexible disc); two flexible disc drives; a flexible disc drive and a memory volume.
2. Copy all modules except LAST from the source INITLIB file onto the destination disc. (You may also remove modules from it as it is copied, if desired.)
3. Add the RS232 module to the INITLIB file on the destination disc.
4. Copy the module named LAST from the source INITLIB to the destination file.
5. Replace the existing INITLIB with the new file.

Mass Storage Requirements

As shipped, the INITLIB file requires about 750 sectors (190 Kbytes) of disc space. Since you will have two copies of this file on-line, and one cannot be removed during the process of adding modules to it, here are the mass storage requirements:

- If you are using small-capacity flexible disc drives (approximately 270 Kbytes per disc), then you will need either two drives or one drive and a memory volume.
- If you are using an SRM shared disc, or have a local hard disc (all have volumes with capacities of 1 Mbyte or greater) or an SS80 flexible disc (approximately 630 Kbytes per disc), then you will need only one disc.

You may also need to initialize one or two blank discs (with the MEDIAINIT.CODE utility) then label them. Disc initialization is discussed in the *Pascal User's Guide*. Write the name on a label before applying the label to the disc; sharp instruments are likely to damage the disc. You will probably also want to make a back-up copy of the BOOT: disc on which the INITLIB file resides.

Making a Memory Volume

If you only have one small-capacity flexible disc drive, then you will need to make sure that you have enough memory to make a memory volume of sufficient size. Here is how to determine the amount of memory in your computer. If the machine is on, turn it off, along with any disc drives to which it is connected. Open the doors of any built-in flexible disc drives. If the SRM is already connected, remove the cable connected to the 98629A Resource Management interface in the back. Now turn the computer on. After going through self-test, the CRT will display the amount of available memory. If you have at least 524 000 bytes, there is "enough" memory to proceed with only one small-capacity disc drive. After turning off computer power, you can reconnect cables and then turn on any peripherals connected to your computer.

If you determined that you have "enough" memory and must use some memory for mass storage, the following steps are necessary.

1. At the Main Command Level, press [M]. The computer responds:

```
*** CREATING A MEMORY VOLUME ***

What unit number?
```

2. Enter:

```
#50
```

   The computer then asks:

```
How many 512 byte BLOCKS?
```

3. Enter:

```
520
```

   The computer asks:

```
How many entries in directory?
```

4. You answer:

```
8
```

   The computer finishes:

```
#50:  (RAM:) zeroed
```

5. Use the Filer's Change command to re-name the memory volume from RAM: to DEST: (which is the volume name assumed in the following procedure).

This has reserved 266 240 (520*512) bytes of memory to use as a mass storage device. It is like having a 5.25-inch disc drive with a disc named DEST: inserted in it.

Assumptions Made During this Procedure

Now you are ready to start the process of making the new INITLIB file. This procedure makes the following assumptions:

- The existing INITLIB file is on the BOOT: disc.
- The destination volume is called DEST:.
- The RS232 driver module is on the CONFIG: disc.

The Procedure

1. Make sure that the Librarian is on-line (or insert the ACCESS: disc into the drive you have been using) and press [L] to load it.
2. When you see the Librarian's prompt line at the top of the CRT, press [O] to specify the name of the (Output) file the Librarian will be creating. Enter this name as the destination (new library's) file name:

   ```
   DEST:INITNEW
   ```

   Note

   If you are using a flexible drive, you must not remove the Output disc until the end of this procedure, after you have Quit the Librarian.

3. Press [I] so you can specify an Input file, then enter:

   ```
   BOOT:INITLIB.
   ```

   If the file is not on the BOOT: volume, then you will need to change the leading volume specification. Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the Input file.
4. Near the bottom of the screen you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press [T] to transfer this module to the output file. After a few moments, the name of the next module will appear (probably SYSDEVS). Each time a new module name appears, press [T] to transfer it to the Output file. You should continue copying modules until the name LAST appears. **Don't copy module LAST yet.**
5. Now you must get the required RS232 drivers from the CONFIG: disc and transfer them to the Output file. If the CONFIG: disc (or a disc containing the RS232 module) is not currently on-line, then put it on-line. Press [I] for an Input file and enter this file specification:

   ```
   CONFIG:RS232.
   ```

   (If you are not copying the module from the CONFIG: disc, then use the volume specification of your source disc.) Don't forget the period after the file name.
6. When the module name RS232 shows up near the bottom of the screen, press [A] which tells the Librarian to transfer All the modules in the file. Remove the CONFIG: disc after the module has been transferred.

7. Put in BOOT: once more, press [I] for Input, and enter the file specification of the original INITLIB file:

   ```
   BOOT:INITLIB,
   ```

8. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing [M] for module and enter:

   ```
   LAST
   ```

   Then transfer LAST to the Output file by typing [T].

9. You now have all the modules in your new library. "Keep" it by typing [K].

10. You may want to verify that these modules are in the new library. Press [I] and specify the new library as the Input file:

    ```
    DEST:INITNEW
    ```

    (The .CODE will automatically be appended to the file name.) Step through the file with the space bar. If all modules are present, then Quit the Librarian by typing [Q].

11. Remove the old copy of INITLIB from the BOOT: disc with the Filer's Remove command (if the Filer is not on-line, you will need to put it on-line before trying to invoke it). Then Krunch the BOOT: disc so that you will have enough room on the disc to store the new, larger copy of the INITLIB file (INITNEW.CODE).

12. Use the Filer's Filecopy command to copy the new library file (DEST:INITNEW.CODE) onto the BOOT: disc, changing the name INITNEW.CODE to INITLIB as you copy it.

13. The next time that you boot your system with this new INITLIB, module RS232 will automatically be installed.

Modifying the TABLE Program

This section first describes the structure of the TABLE program. If you want to change something that it does, you will need to edit and re-compile the CTABLE.TEXT source program on the CONFIG: disc.

Overview of General Steps

Here are the general steps you should take to modify TABLE:

1. The Pascal source of TABLE, called CTABLE.TEXT, is provided on the CONFIG: disc distributed with every copy of the system. Read the commentary on the CTABLE source program in this section. You should follow along in the source program as you read the corresponding commentary.
2. Make your modifications to a *copy* of the program – **not the original.**
3. Compile this modified program, which yields an object code file (for instance, MYTABLE.CODE).
3. Execute the modified program to see if the results are correct. In fact, the Unit Table can be reconfigured any time by executing a version of TABLE.
4. When you are quite sure the new TABLE program is correct, use the Filer to copy the compiled code to your BOOT: disc. The name of the copy must be TABLE (not TABLE.CODE) in order to be recognized during boot-up (with Boot ROM 3.0 and later, it may begin with the letters TABLE and end with letters that match the other BOOT: file names; see the discussion of Renaming the BOOT: Files earlier in this section).

CAUTION

BE CAREFUL HERE! IF THERE IS NO BACKUP COPY OF THE BOOT: DISC CONTAINING THE ORIGINAL TABLE, AND THERE IS SOMETHING WRONG WITH THE MODIFIED TABLE, YOU MAY NOT EVEN BE ABLE TO USE YOUR SYSTEM.

---

5. Depending on the size of INITLIB, there may not be much room on the BOOT: disc. You may need to Krunch the disc with the Filer to make space. The modified TABLE can also be made considerably smaller by linking it to itself. This combines all the internal modules into a single module, and gets rid of module interface text and internal reference information. The procedure for linking the modules of a file is presented later in this section.

Commentary on the CTABLE Program

CTABLE is a long program; for ease of study, here is a summary of its structure. You will probably want to print out the source code and examine it in detail.

```
program ctable;

    module options;
        {Contains declarations which MAY BE EDITED to override
        many of the system defaults.}

    module ctr;          {DON'T MODIFY THIS MODULE.}
        {Exports the table entry assignment routines, which contain
        information highly specific to HP peripheral devices.}

    module BRstuff;      {DON'T MODIFY THIS MODULE.}
        {Figures out which device was the boot device.}

    module scanstuff;    {DON'T MODIFY THIS MODULE.}
        {Contains code which asks each device to identify itself.}

begin

    initialize hardware (interfaces, etc.).

    assign default 'device address vectors'.

    scan for devices on various HPIB addresses.

    scan for an internal mini-floppy drive.

    determine the nature of the boot device.

    create temporary unit table.

    make 'standard' assignments #1:-#6:
      (in temporary Unit Table).

    assign units #7:-#10: (to 2nd and 3rd priority floppies).

    assign units #11:-#40: (to local hard discs).

    assign units #41:,#42: (to tape drives).

    assign units #43:-#44: (as alternate DAMs for #3:-#4: floppies).

    assign unit #45: as additional entry for SRM
       (note the template for #46).

    assign units #47:-#50: (as alternate DAMs for #7:-#10: floppies).

    make optional templates for 'manually' overriding preceding defaults
      [hard-disc partitioning, tape drives, EPROM and Bubbles, etc.].

    copy temporary Unit Table to actual system Unit Table.
```

```
    prefix directory on SRM for #5: (default) and #45: (system),

    remove extraneous local hard disc entries if necessary,

    assign unit for system volume,

    set prefix of SRM default volume,

    re-open the 'standard' system files (#1:, #2:, and #6:),

end,
```

Note

You may want to read through the following discussion of the standard TABLE in its entirety before editing the CTABLE.TEXT source file. On the other hand, you may want to edit it as you read the discussion.

The general recommendation is that you should edit the main program **only** if the desired result cannot be obtained by modifying the declarations in module `options`.

Modifying Module OPTIONS

This module consists only of declarations of exported types and constants. The constants are used by the main program; each section describes their effects and how to modify them.

Power-Up System Volume

The following constant selects the system volume at power-up.

```
{power-up system unit}
  const
    specified_system_unit =
      0; { <>0 overrides auto-assignment}
```

When `specified_system_unit` is zero (the default), the program makes its own choice according to the algorithm described in the preceding discussion of The Booting Process.

If you change this constant to a non-zero value, then it indicates which of the 50 units is to be the system volume. For instance, if you change this constant to 3, then drive #3: will become the system volume. This explicit choice overrides any units specified in the subsequent `{ system unit auto-search declarations }` section of this module.

Primary Directory Access Method (DAM)

The following constant selects the primary Directory Access Method for *local* (i.e., non-SRM) mass storage devices.

```
{local mass storage directory access method}
  type
    lms_dam_type = {local mass storage dam}
      ( LIF, UCSD );
  const
    primary_lms_dam =
      LIF;
```

`LIF` selects HP's Logical Interchange Format directory; `UCSD` specifies the default format used in Pascal 1.0. The standard TABLE expects that remote mass storage devices will use the Shared Resource Manager's hierarchical (structured) directory format (SDF), so no constant declaration is needed.

Floppy Disc Unit Pairs

The standard TABLE program is set up to assign unit numbers to up to three dual floppy disc drives. They will normally occupy units in pairs (#3: and #4:, #7: and #8:, and #9: and #10:). Hard discs usually begin at unit #11: and can be assigned up to 30 unit numbers (up to unit #40:).

```
{floppy/harddisc unit number slot tradeoff's}
  const
    floppy_unit_pairs =  {[1..10]}
      3;
    harddisc_first_lun = {do not edit!}
      7+(floppy_unit_pairs-1)*2;
    harddisc_last_lun =
      40;
```

In order for the TABLE program to assign unit numbers to other than three floppy-disc pairs, you will need to change the `floppy_unit_pairs` variable accordingly. Note that changes to this constant affect the beginning unit assigned to the highest priority hard disc in the system. For instance, changing the constant to 2 causes hard discs to begin at #9, while changing it to 4 causes hard discs to begin at unit #13.

Local Printer Type Option

This constant determines the type of the local printer; it can be either HPIB or RS232.

```
{local printer type option}
  type
    local_printer_type = (HPIB, RS232);
  const
    local_printer_option = HPIB;
```

Local printers with RS-232C interfaces will **not** be recognized by the standard TABLE program. If option `RS232` is chosen, you must have an HP 98626 or 98644 RS-232C Serial interface or an HP 98628 Datacomm interface present. In order to use one of these Serial interfaces, module RS232 must be installed; using the Datacomm interface requires module DATA_COMM.

Here are the default interface switch settings:

Select code 9
Interrupt Level set to 3
Baud rate set for 2400 baud
Stop bits set for 1 stop bit
Bits/character set for 8 bits
Protocol set for XON/XOFF
Parity set to off

If your printer uses a different parameter, then you should change the interface switch setting[1] accordingly; see the interface's installation manual for switch locations and settings. If you want to use another select code, then you will need to modify the `local_RS232_printer_default_dav` parameter accordingly; see the subsequent discussion of Default Device Address Vectors.

Local Printer Timeouts

This constant determines the timeout parameter for local printers:

```
const
  local_printer_timeout =
    $IF local_printer_option=HPIB$
       12000;  {milliseconds}
    $IF local_printer_option=RS232$
       0 ;     {infinite}
```

This governs the byte-transfer timeout used by the local printer driver. The timeout, expressed in milliseconds, specifies the maximum time allowed for each byte handshake to complete. A value of zero is a special case, specifying an infinite timeout. See the commentary above this constant declaration in the CTABLE.TEXT source program for recommended values.

The policy of enforcing a timeout on each individual byte works quite well with most HP-IB printers, since they tend not to hold off bus handshakes much longer than the time it takes them to print a single character. However, with printers on other interfaces (notably serial interfaces) we have a different matter. Some serial printers will "buffer up" bytes at high speed until their internal buffer is full, but then will not allow any more tranfers until their internal buffer is almost empty. Thus, depending upon the printer's internal buffer size, the maxmimum time between two bytes being transferred may be the time it takes to print hundreds or even thousands of characters! For these printers, you might consider a timeout of several minutes, or even an infinite timeout.

In general, most HP-IB printers accept hundreds of bytes per second, so you might think that the default 12 second timeout is excessive. We were forced to use this large a number since some low-cost HP-IB printers take 8-10 seconds to execute a full-page formfeed. If you are using a faster printer, you might consider reducing the timeout to 2-3 seconds, so that a real timeout condition will be detected more quickly.

1 With the 98644 card, you may need to write a short application program to set the parameters. See the *Pascal Procedure Library* manual for details.

Default Device Address Vectors

Here are the default "device address vectors" for devices that cannot be found by interrogation.

```
{default dav's for devices not found by scanning}
  type dav_type = {device address vector}
    packed record
      sc, ba, du, dv: -128..127;
    end;
  const
    HP9885_default_dav =
      dav_type[sc:12, ba:-1, du:0, dv:-1];
    SRM_default_dav =
      dav_type[sc: 21, ba: {node} 0,
               du: {unit} 8, dv: -1];
    BUBBLE_default_dav =
      dav_type[sc: 30, ba: 0, du:  0, dv:  0];
    local_HPIB_printer_default_dav =
      dav_type[sc:  7, ba: 1, du: -1, dv: -1];
    local_RS232_printer_default_dav =
      dav_type[sc:  9, ba: 1, du: -1, dv: -1];
```

The device address vector, or dav, is the data type which describes how a peripheral device is addressed. These constants set up the addressing which is normally used to talk to some standard peripheral devices (some of the information will be overridden if the peripheral is found at a different address).

- `sc` is the interface select code. Select code 7 corresponds to the built-in HP-IB port at the back of Series 200 computers. The HP 9885 disc is connected using a 16-bit parallel interface on select code 12, and a DMA card. The SRM interface is normally set to select code 21.
- `ba` is the HP-IB primary address of the peripheral. Usually an 8290x is addressed as device 0; so is a 9895. The 913x family of hard discs are expected (though not required) to be at primary address 3, and printers at address 1.

  For the SRM only, `ba` indicates the node number of the SRM interface in a cluster (as opposed to the node number of the Workstation itself).
- `du` selects the disc unit in a multi-drive machine. For instance, a 9121D has drives 0 and 1. With SRM systems that contain multiple disc drives, this parameter selects which disc is to be accessed.
- `dv` selects a particular volume in a multi-volume CS80 (7908 family) disc.

Hard Disc Partitioning

The next section that you will come to in the CTABLE.TEXT source program concerns hard disc partitioning. However, in order to be able to wisely decide whether or not you will need to modify any of these parameters (and, if so, to choose which parameter you want), you need to fully understand how partitioning works.

Note

If you haven't read the discussion of hard disc partitioning in The Booting Process discussion, you should do so now.

The standard TABLE program assumes that local hard discs are to be partitioned into several "logical volumes," each of which is to be assigned a unit number. The following equation conceptually describes how TABLE determines the number of volumes into which the disc is to be logically divided (the actual equation actually used is slightly more complex, because it partitions on track boundaries):

`nvols` = disc capacity DIV `mvs`

The values of `nvols` for each type of hard disc, as calculated by this equation, are shown near the end of the main part of the CTABLE program, following the comment:

```
{ templates for "manually" specifying mass storage table entry assignments }.
```

The `mvs` parameter is a constant in the `options` module.

```
{local hard disc partitioning parameters}
  type
    pp_type = {partitioning parameters}
      record
        mvs: integer;   {min vol size in bytes}
        mnv: shortint;  {max number of volumes}
      end;
```

<Comments in program about values and effects of `mnv` parameter.>

```
  const
    min_size = {in bytes [1..maxint]}
      1000000;
    max_vols = {[-30..30]; <0 means auto-coalesce}
      -30;
    HP913X_A_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    HP913X_B_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    HP913X_C_pp =
      pp_type[mvs: min_size, mnv: max_vols];
    CS80disc_pp =
      pp_type[mvs: min_size, mnv: max_vols];
```

The constant min_size indicates that no logical volume is to be smaller than one million bytes. The constant max_vols indicates that no device is ever to be partitioned into more than 30 logical volumes; the negative value of max_vols indicates that logical volumes that do not have valid directories are to be "coalesced" with the last preceding volume found to have a valid directory. These constants are assigned to the mvs and mnv constants for each class of device. You can change them if desired; the values and corresponding effects of mnv are described in the comments in the CTABLE program.

Note

The constant max_vols must **not** be greater than 30.

The HP913X_A corresponds to all 5-Mbyte HP 913x Option 10 "A" drives and "V" drives with a single "disc unit" or "drive number" (as opposed to non-Option 10 "A" drives which have 4 drive numbers).

The HP913X_C corresponds to all 15-Mbyte HP 913X "XV" drives.

Example of Standard Partitioning

In order to better understand partitioning, let's look at how the standard TABLE program partitions an HP 7908 hard disc. You can see most of the default parameters by looking in the templates section of the main program that begins with this comment:

```
{ current CS/80 discs "soft" partitioned by the host }.
```

These discs have a capacity of about 16 Mbytes, so nvols is 16 for this type of disc.

The size of each logical volume is given by this equation:

```
vol_bytes = tpm DIV nvols * bpt
```

in which:

vol_bytes = size of volume (in bytes)

tpm = number of tracks per disc media (device-dependent)

nvols = number of volumes expected on the disc (device-dependent)

bpt = bytes per track (device-dependent)

The last volume on the disc may contain some additional bytes according to the remainder of the above integer division:

Last volume = vol_bytes + (tpm MOD nvols) * bpt

Here are the values of the preceding parameters for an HP 7908 disc drive (they are contained in the main body of the program, near the end):

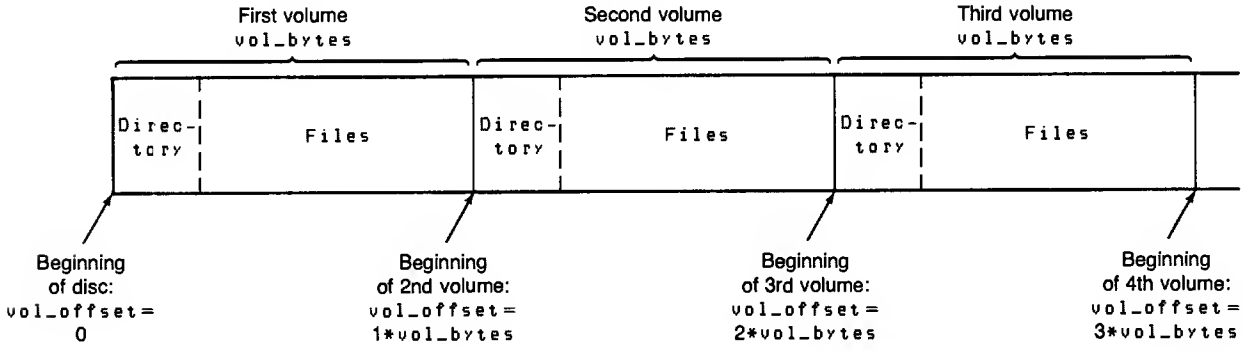
```
tpm   = 5 * 370 { 5 surfaces with 370 tracks/surface }
nvols = 16
bpt   = 35 * 256 { 35 sectors/track with 256 bytes/sector }
```

Therefore:

```
vol_bytes = ((5 * 370) DIV 16) * 35 * 256
          = 1 030 400 (bytes)
```

The `tpm`, `bpt`, and `nvols` parameters for 913x hard discs are found in the `medium_parameters` function in module `ctr`.

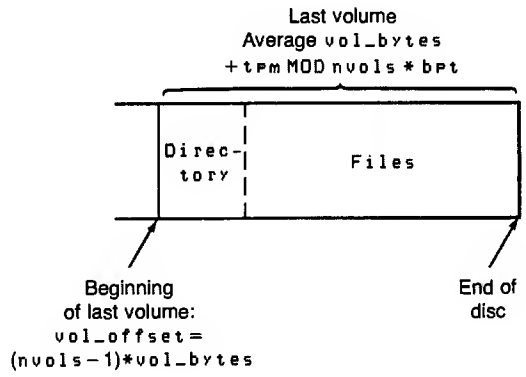
Here is a diagram of how the TABLE program partitions hard discs:



The first directory is placed at the "beginning" of the disc (at an offset of 0 bytes on track 0). The data area used for files immediately follows the directory.

The next directory is placed so as to just follow the end of the first volume. The size of the first volume determines the actual location where the second logical volume will begin. This rule is also followed by each successive logical volume on the disc.

The last volume on the disc looks as follows:



When TABLE attempts to validate unit numbers, it looks at these logical volume boundaries in search of valid directories. As each valid directory is found, the corresponding volume is assigned a unit number. If a valid directory is not found at its expected location, then the corresponding unit is invalid; the amount of disc space normally occupied by this volume can be coalesced with the last preceding logical volume found to have a valid directory, if desired.

Partitioning Recommendations

Here are the general recommendations as to how you can change the standard partitioning of hard discs:

1. The simplest method of changing the partitioning on your hard discs is to "coalesce" adjacent logical volumes. You should try to use this solution if possible.
2. If coalescing does not provide you with an adequate solution, you can also set up your own logical volume structure on the disc by modifying the parameters in the CTABLE source program.
   a. The easiest changes might be to change the `nvols` parameter in the `templates` section that corresponds to your disc. For instance, changing this constant from 30 to 15 for the 7912 allows you to have two 7912 drives automatically assigned unit numbers by CTABLE. The size of the logical volumes will be doubled, and partitioning will still be made on track boundaries.
   b. If the above methods still do not provide an adequate solution, read the subsequent discussion in Designing Your Own Partitioning Schemes.

Coalescing adjacent hard disc volumes was discussed earlier. Modifying the TABLE Program is discussed momentarily.

Note

If you do modify the standard TABLE program, keep in mind that you must use a version of the program that uses the *same* partitioning scheme in order for all logical volumes to be recognized properly.

Designing Your Own Partitioning Schemes

The most highly recommended method is the standard TABLE partitioning method. Here is the section of the `template` (toward the end of the main CTABLE program) that performs the standard partitioning:

```
for i := 0 to nvols-1 do
  tea_CS80_mv(11+i, primary_dam, {sc} 7, {ba} 0, {du} 0, {dv} 0,
                    vol_offset(i, nvols, mp),
                    {devid} CS80id,
                    vol_bytes(i, nvols, mp));
```

The vol_offset and vol_bytes functions calculate the offset and size of each of your directories according to the nvols and mp values; you can use the standard values, provided in this same template, or specify your own. If, for example, you wanted to change only the value of nvols for a 7908 disc to 8, you could change this line in the template (just a few lines before the standard partitioning algorithm shown above):

```
CS80id := 7908; nvols := 16; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

to this:

```
CS80id := 7908; nvols :=  8; mp.tpm := 5* 370; mp.bpt := 35*256; {7908};
```

The vol_offset and vol_bytes functions would then make the volume offset and size calculations for you.

While the standard method is the most highly recommended, there is nothing that prevents you from using your own. If you like, you may remove the for statement, duplicate the tea procedure call n times, and specify volume offsets and sizes of your choosing for each logical volume. Here is an example of one for unit 11 (you will have to supply actual values of the example parameters offset_for_unit_11 and bytes_for_unit_11 shown below):

```
tea_CS80_mv(11, primary_dam, 7, 0, 0,
                  offset_for_unit_11,
                  CS80id,
                  bytes_for_unit_11);
```

The tea procedure checks to ensure that your logical volumes each lie inside the media boundaries. Unfortunately, the tea procedure doesn't check to see if any of them overlap!

In those templates capable of partitioning media, you will find the following line:

```
{ mp := block_boundaries(mp); {override track boundary partitioning}
```

This allows you to use the standard partitioning method, except that the partitioning will occur on 512-byte block boundaries – not necessarily on track boundaries. The "{" character at the beginning of the line makes the line a comment; enable compilation of the line by deleting the "{" character. Depending upon the media parameters and the number of logical volumes, this may or may not make a difference in how your media actually gets partitioned. This feature is provided solely for compatibility with discs used with Pascal 1.0. If you don't need it for this reason, don't use it!

All parameters in the templates have typical values for your convenience. If you get a "value range error" when you execute your modified version of CTABLE, it probably means that one or more of your parameters is out of range. Don't worry about your system configuration; the old configuration will still be in effect. You can immediately go back to the Editor to try to determine the problem with your new CTABLE.

To find where the value range error occurred, usually the quickest way is to examine the tea procedure calls you just modified, and then examine the tea procedure itself to see what range it checks the parameters for. However, unless you are a certified wizard, don't modify the tea procedure itself!

If you still can't find the source of the error, you can re-compile CTABLE with $DEBUG ON$. Get a listing from the Compiler, too. Then execute CTABLE again. When it terminates with the error again, use the queue (Q) command in the Debugger to determine the line numbers of the statements leading up to the error. Also, when you examine the queue, you may need to trace back several line numbers to actually locate the offending statement.

System Unit Auto-Search Declarations

These constants determine the order of devices searched while trying to find a system volume.

```
{system unit auto-search declarations}
  const
    sysunit_list_length =
      7;
  type
    sysunit_list_type =
      array [1..sysunit_list_length] of unitnum;
  const
    sysunit_list =
      sysunit_list_type[
      harddisc_first_lun, {first hard disc logical unit number}
      45,   {srm, prefixed to user's sysvol}
      4,    {floppy unit 1, primary dam}
      44,   {floppy unit 1, secondary dam}
      3,    {floppy unit 0, primary dam}
      43,   {floppy unit 0, secondary dam}
      42];  {bubbles}
```

If a valid directory is not found on any of these units, then the system volume is determined by the normal algorithm (described in The System Volume section of The Booting Process discussion presented earlier this chapter).

If a system unit was expilcitly specified by modifying the constant called `specified_system_unit` at the beginning of the module called `options`, then this search will not override the specified system unit.

HP-IB Select Codes Searched

These constants determine the select codes scanned in search of an HP-IB type interface, including 98625 High Speed Disc interfaces.

```
{HP-IB select code scanning declarations}
  const
    sc_list_length =
      3;
  type
    sc_list_type =
      array [1..sc_list_length] of shortint;
  const
    sc_list =
      sc_list_type [
        7,      {internal HP-IB}
        8,      {default sc for HP98624 HP-IB}
        14];    {default sc for HP98625 HP-IB}
```

The select codes are searched in the order they appear in the list (7 first). On each select code, addresses 0 through 7 are polled in succession for devices. In the case of multiple devices contending for an assignment class, say multiple local hard discs where the total capacity of all is greater than 30 Mbytes, generally the last one polled will be the one assigned a logical unit number.

About Module CTR

This module should not be modified!

Built into it is a lot of knowledge about the supported HP mass storage products, and provides a general structure into which can be inserted information about new peripherals as they are introduced.

Each peripheral is assigned a letter designator; these are listed in the export section of module `ctr`. In addition there is descriptive information about the size of each type of device, expressed in bytes per track and tracks per medium. The routines in `ctr` avoid partitioning across track boundaries, which would cause very inefficient disc access patterns.

Most of the procedures exported from `ctr` are given a name prefixed with `tea_` . These are the Table Entry Assignment routines. There are `tea` routines for all the supported mass storage products. Some `tea` routines are appropriate for an entire family of related mass-storage products.

There are also some utility routines. The `create_temp_unitable` procedure allocates in the heap a temporary structure like the real system Unit Table. CTABLE makes its assignments to this temporary structure, then uses `assign_temp_unitable` to copy the final result into the actual system table. Note that `assign_temp_unitable` will not overwrite any RAM volumes which have been created in the system unit table. This feature is provided so that if you execute a CTABLE while the system is running, you won't lose files in memory.

The `sysunit_ok` function checks to see if a particular unit is blocked, on-line, and has a valid directory; if so, it is a legal candidate for the system unit.

If you look at the assignments to the various fields of a Unit Table entry, you will be aware that two of them are procedure variables which must be initialized to the names of the DAM (Directory Access Method) and TM (Transfer Method or driver) appropriate to the volume and physical device. DAMs and TMs are not part of CTABLE and so would ordinarily be linked to modules already in RAM by the linking loader when CTABLE is loaded.

However, there is no guarantee that the DAMs and TMs for a device are present, since they may have been removed from INITLIB or never even installed. Consequently, CTABLE has been programmed to examine the symbol tables kept in memory by the linking loader. If a driver's name is found, it can be used; otherwise, the program avoids references to absent drivers. The routine which searches for link symbols at run-time is called `value` and is exported from module `ctr`.

About Module BRSTUFF

This is another module which shouldn't be modified!

It exports two routines. The function `internal_mini_present` determines if there are any internal flexible disc drives in your computer. The function `get_bootdevice_parms` determines what type of device was used for booting and returns the `dav` (device address vector) for that device.

About Module SCANSTUFF

This module shouldn't be modified!

Its purpose is to interrogate certain disc drives about their size and identification. To do this, the `value` routine (see module `ctr`) is used to find routines which are present only if the driver modules supporting these discs are installed.

Discussion of the Main Body of CTABLE

A lot of details of the behavior of CTABLE can be modified by changing declarations such as the select code list from the options module. If you want to force some particular assignment, this may be achieved by modifications to the code in the body of CTABLE.

Default DAV Assignments

The program first assigns default device address vectors (DAVs) for devices that cannot be found by scanning (such as printers and HP 9885 8-inch disc drives).

HP-IB Interfaces Scanned

After various initializations, CTABLE scans the select codes listed in module options. For each HP-IB interface found, and for bus addresses 0 through 7 on each interface, the program inquires to see if a device is present. A letter designating the device is returned. You can see the definitions of these letters in the constant declaration at the beginning of the `ctr` module.

Boot Device Info

The information about the boot device is obtained. This may be used later in selecting the system unit.

Temporary Unit Table

A temporary Unit Table is then created in the heap. The assignments made as CTABLE executes will be made to elements of this temporary table; only at the end will the real system Unit Table be updated.

Standard Assignments

Next, "standard" unit number assignments are made. It is wise not to change these assignments, since programs tend to depend on them.

- Unit #1 is assigned to the screen (CONSOLE:)
- Unit #2 is assigned to the keyboard (SYSTERM:)
- Units #3 and #4 will be assigned to the highest priority flexible disc drive. If both internal drive(s) and an external flexible disc drive are present, the internal drive(s) will be used for #3 and #4 unless the external disc was the boot device. This policy gives preference to the higher-performance internal floppy disc drives.
- If an SRM interface is present, it is assigned unit #5. (It may also be assigned unit #45 later in the program.)
- Unit #6 is assigned to the local printer (PRINTER:). This assignment is made whether or not a printer is actually connected to the computer, because there is no way to interrogate every possible type of printer.

Additional Floppy Unit Pairs

Next, the second and third pair of flexible disc drives are assigned unit numbers. Units 7 and 8 are assigned to the second highest priority floppy drive pair, and 9 and 10 to the third priority pair.

Multiple Local Hard Discs

With auto-configuration, CTABLE can deal with several local hard discs found during the HP-IB scanning process (previous versions of this program, without modification, could only find one). This code is surrounded by conditional Compiler options, because you may wish to not compile it and instead force particular assignments.

CTABLE will break a hard disc (which has not previously been initialized to a single volume) into multiple volumes. As things are arranged (see module `options`), no volume will be less than one million bytes and no disc will be divided into more than 30 volumes. The units assigned to these volumes begin with #11 and can use up through #40, depending on the number required for each disc.

DC600 Tape Drives

If there are any DC600 stand-alone tape drives present, or any CS80 disc drives with integrated tape drives present, then the program also finds them. The highest priority tape is assigned unit number 41, and the second priority tape is assigned unit number 42.

Alternate DAMs

Next, the alternate-DAM entries are assigned. This allows all flexible discs to be used regardless of the resident directory type. Units #43: and #44: are alternates for #3: and #4:. For instance, if LIF is the primary DAM, then units #43: and #44: will use the alternate UCSD DAM to access units #3: and #4:. (Alternates for units #7:-#10: are a few lines later in the program.)

Duplicate SRM Unit Entries

The "duplicate entries for prefixing down the SRM" section provides templates that you can use to assign additional unit numbers to SRM directories. For instance, suppose you want to have unit #46: assigned to the directory called `/SPECIAL/USER10/FRED`. Enable the first template by deleting the `{` comment brace preceding it. Then scroll down until you find the comment `{ prefix the primary and secondary SRM unit entries }`. (It may be easier to use the Editor's Find command, since these templates are a couple of pages away from the first templates.) Enable the template for #46: by deleting the `{` comment brace, and replace the `?` with the desired directory path `SPECIAL/USER10/FRED`.

#45 is not really an alternate; it is another SRM volume, and may be assigned as the system volume later. If this happens, the operating system will have two units on the SRM: one for the "system volume," which is used for temporary system files, work files, stream files etc.; and another for the "default" working directory. This avoids any possible need to prefix an SRM system volume away from an SRM default volume.

More Alternate DAMs

Next, units #47: and #48: are assigned as alternate DAM units for #7: and #8: (second priority floppy disc pair). Units #49 and #50 are alternates for #9: and #10: (third priority floppy disc pair).

Templates

Next are the "templates" for overriding the mass storage table entry assignments made by the standard TABLE. These templates are surrounded by conditional `$if false$` Compiler options which cause them to be skipped. Thus, the `tea` procedure calls have no effect until you change the `$if false$` to `$if true$`. The `tea` procedures themselves, are defined in the module `ctr`. They actually perform the Table Entry Assignments.

There are templates for the following disc drives: internal; 8290x (Amigo); 9895; 913xA, B, V, and XV; SS80 flexible discs (such as the 9122) and CS80 hard discs (HP 7908, 7911, 7912, 7914, 7933, and 7935); CS80 tapes; EPROM cards; Magnetic Bubble memory cards. Each template gives the opportunity to specify the following:

- directory access method (DAM)
- select code
- bus address (HP-IB interfaces)
- drive unit
- offset in bytes from beginning of volume to this unit's directory (for multi-volume discs)
- drive type (the variable named letter in a constant declaration of module `ctr`)
- size of volume

For multiple-volume drives, the templates include a `for` loop which calculates how to break up the disc space in the preferred fashion.

If you want to change the default for an HP 9121 drive, you will need to use the HP8290X `tea` procedure. The reason for this is that the HP 9121 drives behave just like the HP 8290X drives. You might also note that you would also use the HP8290X `tea` procedures for the 5.25-inch drive in the HP 9135 and the 3.5-inch drive in the HP 9133.

The first parameter in the `tea` procedures specifies the unit number you wish to assign. It must be in the range from 1 thru 50. The second parameter specifies the directory access method, or DAM. The DAM specifier is of emumerated type "ds_type". Exported from module `ctr`, ds_type is shown here.

```
type
  ds_type = {Directory access method Specifier for local mass storage}
    ( primary_dam,      {either LIF or UCSD, as specified in options}
      secondary_dam,    {the one not selected as primary}
      LIF_dam,          {LIF, regardless of primary/secondary choice}
      UCSD_dam  );      {UCSD, regardless of primary/secondary choice}
```

A `tea` procedure has parameters only for those items which are applicable to the device. Furthermore, all parameters are range-checked by the `tea` procedure. While the range-checking cannot guarantee the correctness of your parameters, it can nearly guarantee that your parameters won't ruin the system.

The remaining parameters for all the local mass storage `tea` procedures are device-specific. Most devices will need addressing information such as select code (`sc`), HP-IB bus address (`ba`), and disc unit number (`du`).

You may leave the templates where they are, or you may move them. However, all `tea` procedure calls must take place between these two statements:

```
{ Create a temporary table & fill it with dummy entries }
create_temp_unitable;
```

Place all `tea` procedure calls here.

```
{ assign the new unitable and unitclear all units }
assign_temp_unitable;
```

You may assign and re-assign logical units as many times as desired between the two statements above. When the same logical unit is assigned multiple times, the last assignment performed will be the one that remains in effect.

Temporary Unit Table Copied

Next, the temporary unit table is copied into the system's unit table (except that RAM volume entries are not overwritten).

SRM Prefix Directories

The SRM unit entries are then prefixed to the appropriate directories. Each workstation in an SRM system has an identification number called its "node number", and it is **strongly** recommended that the system be configured so that every workstation's node number is unique.

CTABLE tries to prefix #45 to a directory called /WORKSTATIONS/SYSTEMnn, where nn is the node number. If no such directory exists, it tries to use directory /WORKSTATIONS/SYSTEM (with no node number). If that one doesn't exist, entry #45 is nullified. This is a rather key mechanism. It allows the workstations in an SRM system to have unique configurations. For the normal functioning of the Pascal system, a system volume is required to hold the system library and various system files. If all workstations shared the same system volume, file name collisions would be a real nuisance. CTABLE supports this partitioning, and so does the overall booting process, allowing for instance a different INITLIB and TABLE for each workstation.

Remove Extraneous Hard Disc Volumes

When a valid directory is not found at the expected location on the disc, then the corresponding unit number is not valid. This service is performed by the section of code in the main part of CTABLE that follows the comment:

```
{ remove extraneous local hard disc entries if necessary }.
```

If desired, the volumes which don't have valid directories may be "coalesced" with the last valid directory found which precedes this invalid directory.

System Unit Selected

The system unit is then selected according to the priorities set in the constant called `sysunit_list`, exported from module `options`.

SRM System Unit Selected

Finally, if the system unit is #45: (SRM system volume), then unit #5 is also an SRM volume. In that case, #5 is prefixed back to the root SRM directory (`#5:/`) so the root is the initial default volume for the system right after it boots up. You can change the working directory to your own directory by adding the directory path to the slash (/).

System Files Re-Opened

This procedure re-opens the standard unblocked system 'files':

- #1: is assigned to SYSTERM:
- #2: is assigned to CONSOLE:
- #6: is assigned to PRINTER:

Editing CTABLE

If you have just read through the preceding discussion the first time, you will need to go back and read the relevant sections and make the desired changes.

If you have already edited the CTABLE source program, you are ready to store your new file. Quit editing and Write the edited CTABLE in a new file, such as NEWCTABLE (or use the Save option if you are using a backup copy of the file). Exit the Editor by typing E.

Compiling and Running CTABLE

1. The modules in CTABLE.TEXT import modules from INITLIB. However, the interface text for these modules is not available unless you enable the `$search 'CONFIG:INTERFACE'$` Compiler option at the beginning of the source program (by removing the comments from the line). You must also be sure that this disc is on-line during the compilation of the CTABLE program; you could also copy the file onto another on-line disc and change the volume specification in the program accordingly.
2. Load the Compiler by typing (C) (you may need to put the CMP: disc on-line). Answer the Compiler's `Compile what text ?` prompt by entering:

   ```
   NEWCTABLE
   ```

3. Answer the "Printer listing ?" prompt with:

   (Y) for a listing.
   (N) for no listing.
   (E) for an "errors only" listing (if you have a printer).
   (L) for a listing file.

4. Press (Return) or (ENTER) to say that the default output file name of "SYSVOL:NEWCTABLE-.CODE" is fine.

   If you followed the example, you shouldn't have any compilation errors.

5. Press (R) or (RUN) to execute the new CTABLE.

Verifying the New Configuration

Generally, the Filer provides the quickest way to verify your configuration. The Volumes command provides a quick sweep of all units. The List command provides a way to test individual units.

Remember that the Volumes command shows only those units which are on-line and which have valid directories. It won't show units with media containing either no directory or the wrong type of directory.

If the first attempt to List the directory of a unit fails, the Filer displays:

```
Please mount unit #9
'C' coninues, <sh_exc> aborts
```

Type (C). The Filer will then give the reason for failure. A key result is "no directory on volume", which means that the device and medium are accessible, but no directory was found. Other results such as "device absent or unaccessible", "medium absent", or "device not ready" mean that the attempt to read from the device failed.

If you get "device absent or unaccessible", there may be several possible reasons. A good trick at this point is to eXecute ACCESS:MEDIAINIT on the unit number of interest. For those device types MEDIAINIT recognizes, it will print out the expected device type, plus the addressing information. This is an excellent way to verify the expected configuration, even if the device itself is unaccessible. Don't worry about specifying a device that you really don't want to initialize; MEDIAINIT always prompts for your confirmation before it begins initializing.

Making the New Configuration Permanent

Once you are satisfied with your new configuration and wish to make it permanent (i.e., it will remain unless you change it again), copy the code file to your BOOT: disc. First, however, you should link the new file to itself in order to conserve disc space.

Link the Modules Together

1. Invoke the Librarian by inserting the ACCESS: disc and pressing [L].
2. Insert the SYSVOL: disc, press [I] (for Input) and enter:

   ```
   NEWCTABLE
   ```

3. To conserve space on the disc, you can specify a header size smaller than the default (38). Press [H], and enter: `1`. The header size is then changed to the minimum (18).
4. Press [O] (for Output) and enter:

   ```
   NEWCTABLE
   ```

5. Press [L] (for Link).
6. Press [D] (to remove the file's Def table).
7. Press [A] (to link All the modules).
8. Press [L] (to finish Linking).
9. Press [K] (to Keep the file).
10. Press [Q] (for Quit).

Now you are ready to perform the final operations.

Install the New TABLE

1. Insert the ACCESS: disc and type F (for Filer).
2. Remove the original TABLE file. Insert the BOOT: disc, press R (for Remove) and enter:

   ```
   BOOT:TABLE
   ```

3. Krunch the BOOT: disc, since your new TABLE file may be larger than the old one. Press K (for Krunch) and enter:

   ```
   BOOT:
   ```

4. Respond to `Crunch directory BOOT: ? (Y/N)` with Y.
5. Now copy the new code file from SYSVOL: to BOOT:, giving it the required name. Insert the SYSVOL: disc, press F (for Filecopy) and enter:

   ```
   NEWCTABLE.CODE,BOOT:TABLE
   ```

6. Swap discs as directed by the Filer.
7. Save your new source file on the CONFIG: disc too. Insert the SYSVOL: disc, press F and enter:

   ```
   NEWCTABLE.TEXT,CONFIG:$
   ```

8. Swap discs as directed by the FILER.
9. Clean up the SYSVOL: disc by removing all the files you put there. Use wildcards to save typing. Insert the SYSVOL: disc, press R, and enter the ? wildcard.
10. Respond N to the prompt to remove LIBRARY, and respond Y to the prompts to remove INTERFACE, NEWCTABLE.TEXT, and NEWCTABLE.CODE. Respond Y to the confirmation prompt.
11. Exit the FILER by typing Q.

Notes

Example SRM Configuration

The Shared Resource Management (SRM) System is a "file server" system that allows several workstation computers to share file-oriented devices like discs, printers, and plotters. Also, the SRM may be the only mass storage device for a machine with no local disc drives.

This section explains how to configure workstations to access and boot Pascal from an SRM system. It is used as the example "custom" configuration because it can employ three methods of modifying the standard configuration:

- Copying and re-naming files
- Adding modules to INITLIB
- Modifying the TABLE program (optional).

This section tells what to do the *first* time you set up the *first* Pascal workstation to access an SRM system. It should not be repeated for every workstation you set up. Once this procedure is complete, the SRM will be accessible any time you boot up your workstation.

Prerequisites

Here are the assumptions made by this set-up procedure.

Who Should Set Up the SRM

The person who is designated as the "SRM system administrator" should perform the process described in the next few pages.

SRM Hardware

It is assumed that your SRM hardware has been installed and tested as prescribed in the SRM documentation. In order for your system to work with the SRM, every workstation in the SRM configuration must have a unique node number (see the SRM System Manual to learn about node numbers). You will also need the wiring chart and node number assignments which were prepared when designing and installing your SRM system.

SRM 1.0 Operating System Parameters

There are four parameters which are set when the SRM 1.0 Operating System is initially configured. (These are only needed if you are using version 1.0 of the *SRM Operating System*; with SRM 2.0, they are set automatically.) Appropriate values for these parameters when using Pascal Workstations with the SRM 1.0 are as follows:

- IOBUFFERS: At least five per workstation in the cluster – for example, 40 buffers for 8 workstations.
- DISC BUFFERS: Fifty is a good choice.
- TASKS: Two is enough.
- FILES: Allow for ten or twelve open files per workstation in the cluster; one hundred is a nice round number.

Boot ROM Versions

If you have an HP 9816 Computer with a Boot ROM 3.0L, then you must boot from a local disc drive. The SRM can only be used after normal booting is complete. Similarly, if you have an HP 9826 or 9836 Computer with a Boot ROM with version number less than 3.0, then you must boot from the internal 5.25-inch flexible disc drive. In both of these cases, you will probably want to make a back-up copy of the original BOOT: disc, as you will be modifying the INITLIB file on that disc.

If your computer is equipped with Boot ROM 3.0 or later version, it is possible to boot directly from the SRM. System Boot files are found on the SRM system in the /SYSTEMS root directory; they have names like SYSTEM_P. The other files used at boot time (INITLIB, STARTUP, and TABLE) are found in the /WORKSTATIONS/SYSTEM directory. This too is explained in the SRM System Manual.

Overview of SRM Installation

Configuring your system to access SRM is not a hard or complicated operation, but it is important that you follow the subsequent procedures in exact detail. Since you are less likely to make mistakes if you understand what's going on, here is an outline of what you will do.

1. Install driver modules DATA_COMM and SRM by executing them (they are actually programs that install themselves automatically).
2. Execute the TABLE auto-configuration program. When it is executed while the DATA_COMM and SRM driver modules are installed, it will find the SRM system and assign unit #5 to the SRM.
3. If they are not already on the SRM system, create directories /SYSTEMS and /WORKSTATIONS/SYSTEM.
4. Copy the system Boot file (SYSTEM_P) to the /SYSTEMS directory. Copy the rest of the Pascal system files to the /WORKSTATIONS/SYSTEM directory. (The Boot ROM expects to find the Pascal system in these directories.)
5. Use the Librarian to create (on the SRM) a new INITLIB file that contains modules DATA_COMM and SRM, and then replace the existing INITLIB with this new one. (If you have Boot ROM 3.0 or later, then you will be replacing the INITLIB in the /WORKSTATIONS/SYSTEM directory; with earlier Boot ROMs and Boot ROM 3.0L, you will be replacing the INITLIB on the BOOT: disc.)
6. Re-boot the computer, and verify the new configuration.
7. You can also optionally modify the TABLE program to assign additional unit numbers to the SRM system.

Installing the SRM Driver Modules

First, install module DATA_COMM. The file is on the CONFIG: disc that is supplied with your system. Although you may have already copied the file onto another volume, such as a local hard disc, this example assumes that you will be loading and executing it from the CONFIG: disc.

Execute the file by pressing [X] at the Main Command Level. The system prompts:

```
Execute what file?
```

Enter this file specification:

```
CONFIG:DATA_COMM.
```

Be sure to include the trailing period to suppress the ".CODE" suffix.

Install the SRM module similarly; it is also on the CONFIG: disc as shipped to you.

Re-Configuring with TABLE

Use the eXecute command to execute the TABLE program; it is on the BOOT: disc supplied with your system. Press [X], and then answer the `Execute what file?` prompt with this file specification:

```
BOOT:TABLE.
```

Again, be sure to include the trailing period.

When the program has finished, you can use the Filer's Volumes command to see that unit #5 is assigned to the SRM system. From the Main Command Level, press [F] and then [V]. Here is a typical display:

```
Volumes on-line:
  1   CONSOLE:
  2   SYSTERM:
  3 # BOOT:
  5 # ROOT1:
  6   PRINTER:
```

If the name of the SRM's root directory is not shown in the display, re-execute all three programs (DATA_COMM, SRM, and TABLE). You may have done something wrong in that process.

If the Filer's Volumes command still does not recognize the #5: volume, check to see whether the SRM hardware is properly configured and installed. For instance, the (unmodified) TABLE program expects that the SRM interface in your computer is set to select code 21.

If that does not work, then you should refer to the troubleshooting sections of the SRM System Manual.

Creating the Required Directories and Files

The first time that a workstation is set-up to access an SRM system, you will need to set up certain directories on the SRM. These directories have special functions, as described in the following paragraphs.

A Sketch of Normal SRM Directory Configuration

In order to allow each Workstation in an SRM configuration to boot up a unique system and have its own system volume, a private directory is established for each node number.

Strictly speaking, this is not always necessary. If a workstation has a local high-performance mass storage device, then it may be desirable to use that device as the system volume. In fact, the automatic configuration process will select a high-performance mass storage as the system volume, if one is present. However, it doesn't hurt anything to set up unique directories for each workstation. The following discussion explains how to do so. If things are first set up as explained below, you then have the option to copy frequently used files such as the Editor and Compiler from the SRM onto your local high-performance system volume. Then when you boot the system, those files will be found locally and accessed with correspondingly greater speed.

In the SRM's root directory there should be another directory called WORKSTATIONS. Under this there should be a directory called SYSTEM, and for each node number "nn" there should also be a directory called SYSTEMnn. For instance, if there are three Workstations on nodes 08, 14, and 15, then the following directories should exist in the root:

```
WORKSTATIONS/SYSTEM
WORKSTATIONS/SYSTEM08
WORKSTATIONS/SYSTEM14
WORKSTATIONS/SYSTEM15
```

Under WORKSTATIONS/SYSTEM should be copies of all the system files, such as the Compiler, Filer, and Editor.

Under the private directory for each node should be accessible all the files normally used by the Workstation. For files which don't change, such as the Compiler, it is sufficient to simply have a duplicate link to WORKSTATIONS/SYSTEM/COMPILER; there is no need to actually copy such invariant files. The Filer's Duplicate link command can be used for this purpose.

Also in a node's private system directory can be the files which "personalize" a Workstation: customized copies of LIBRARY, INITLIB, AUTOSTART, and so forth.

Once this set-up is created, booting is a smooth and automatic process. With Boot ROM 3.0 and later versions (but not 3.0L), you can boot from the SRM; the particular system to be booted is selected by name at power-up. Thereafter, the Workstation looks for the necessary files in the directory with its node number. If INITLIB can't be found in the /WORKSTATIONS/SYSTEMnn directory, default is taken to /WORKSTATIONS/SYSTEM; if something crucial is still missing, the boot may fail. (The computer will complain to the operator.)

If you boot from the SRM or if you have no local hard disc on-line, your system volume will be unit #45 (prefixed to your private directory /WORKSTATIONS/SYSTEMnn) and your default volume will be #5 (another SRM volume, prefixed to the SRM root directory). Even if the SRM is not chosen as your system volume (using the scheme above), it will still be accessible through units #5 and #45.

In order to run properly, there must be one more special directory called TEMP_FILES under /WORKSTATIONS. All temporary files are created in this directory, and are removed when no longer needed. If you don't create this directory, the first workstation to need it will do so. Should the create fail, an error is reported. Consequently the directory /WORKSTATIONS should not be write-protected unless directory TEMP_FILES has already been created.

Most users will also want a private directory for their default volume. Typically one creates a directory called USERS under the root, and within USERS a private directory for each individual. After booting, use the Filer to set the current working directory for your unit #5 to your private directory (you can modify the TABLE program or create an AUTOSTART file to do this for you). This keeps the root directory from getting cluttered.

Setting Up SRM Directories

Insert the ACCESS: disc in drive #3 and press [F] to execute the Filer. When the Filer prompt appears, press [V] to list the volumes on-line.

If the SRM has already been running with some other systems connected, such as an HP 9845 or 9836 running BASIC, some of these directories may already exist. To see the directories which already exist, press [L] for the List directory command, and enter the root-level directory specification:

```
#5:/
```

In following the steps below, obviously you should skip the steps which create directories which already exist on your SRM.

To create directory /WORKSTATIONS, use the following Filer sequence.

1. Press [M] for the Make-directory command. The Filer responds with this prompt:

```
Make file or directory (F/D) ?
```

2. You want to make a directory, so type [D]. The Filer responds with this prompt:

```
Make directory (valid only for SRM type units)
Make what directory?
```

3. You enter this response:

```
#5:/WORKSTATIONS
```

   Be sure to type this name in capital letters! If the root directory was protected with one or more passwords, the Filer would report: 'Error: invalid password' at this point. In such a case, you need to find out the required passwords from whoever initialized the SRM disc or installed the passwords. To create this directory, you need Write access rights in the root directory, and possibly Manager rights if they were specified. For instance, if the password for Write access is PLEASE, you would specify:

```
#5:/,<PLEASE>/WORKSTATIONS
```

   Alternatively, you might use the main volume password by specifying:

```
#5<VOL_PASS>:/WORKSTATIONS
```

   The Filer should reply:

```
Directory is 'WORKSTATIONS' correct ? (Y/N)
```

4. You answer [Y]. The directory is created, then the Filer announces:

```
Directory WORKSTATIONS made.
```

If the computers in the SRM configuration have Boot ROM 3.0 (or later version) which is able to boot from the SRM, you will also want to create a directory called SYSTEMS in the root. Repeat the steps just given, but instead specify that you want to create directory #5:/SYSTEMS.

Next, create directory SYSTEM under /WORKSTATIONS. This is where the master copy of all system programs such as the Compiler will be stored. To reduce the amount of typing involved, we will make the current working directory for unit #5 be the newly created /WORKSTATIONS directory.

5. Type [P] for the Prefix command. The Filer responds:

```
Prefix to what directory ?
```

6. Enter:

```
#5:/WORKSTATIONS
```

   The Filer will respond:

```
Prefix is WORKSTATIONS:
```

Now if you don't specify a unit number in Filer operations, the system will assume you are referring to directory /WORKSTATIONS. To create SYSTEM, the sequence is as follows:

1. Press M
2. `Make file or directory (F/D) ? D`
3. `Make what directory? SYSTEM`
4. `Directory is 'SYSTEM' correct? (Y/N) Y`
5. `Directory SYSTEM made`

Also under /WORKSTATIONS create directories called SYSTEMnn, where nn is the node number for each workstation in the system. You can see why we said each node number should be unique! For example, create SYSTEM05 for the workstation at node 5. Note that two digits are always required, even if the first digit is zero.

Finally, under /WORKSTATIONS you should create a directory called TEMP_FILES. This is only necessary if you plan to write-protect /WORKSTATIONS.

Copying the System Files to SRM

You are now at the last stage! It is time to move the required files out into the new directories.

1. First prefix the current working directory to SYSTEM. Press P for the Prefix command.
2. Enter this directory specification:

   ```
   #5:/WORKSTATIONS/SYSTEM
   ```

   The Filer responds with this message:

   ```
   Prefix is SYSTEM:
   ```

3. Then insert the BOOT: disc in the drive you have been using and copy all the files on it into the new working directory. Press F for the Filecopy command. The Filer gives this prompt:

   ```
   Filecopy what file?
   ```

4. Specify that you want all files on the BOOT: disc to be copied by using the = wildcard as follows:

   ```
   BOOT:=,$
   ```

   The Filer will copy the files one after another.

Then repeat the above operation for each of the Pascal system discs (ACCESS:, SYSVOL:, etc). After this is done, the /WORKSTATION/SYSTEM directory contains the entire Pascal Workstation system.

Duplicating Links to System Files

Now you need to make these files available in the private SYSTEMnn directory of each workstation. For each such system directory, use the Filer's Duplicate Link command.

1. Press D.

```
Duplicate link (valid only for SRM type units)
Duplicate or Move ? (D/M)
```

2. You want to duplicate links rather than move links. Press D. The Filer will ask:

```
Dup_link what file?
```

3. Answer:

```
?,#5:/WORKSTATIONS/SYSTEMnn/$
```

   Of course you should substitute a two-digit node number for nn each time (a leading 0 is required for single-digit node numbers). The "?" wildcard tells the Filer to ask if you want links each file in the source directory. Answer Y for every file except AUTOSTART and SYSTEM_P.

The Dup_link operation is very fast. It displays each file name as the links are made.

The last detail is optional. If any of the workstations in the SRM system have Boot ROM revisions 3.0 or later and will be expected to boot from the SRM instead of using local mass storage, you need to put a copy of the system Boot file in directory /SYSTEMS (not in /WORKSTATIONS/SYSTEM). The system Boot file (SYSTEM_P) is on the BOOT: disc shipped with the system; you probably have already made a copy of it in an earlier procedure. The Dup_link command can duplicate the file in a different directory.

1. Type D for the Duplicate link command.

   The Filer responds with this prompt:

```
Duplicate link (valid only for SRM type units)
Duplicate or Move ? (D/M)
```

   Respond with D.

2. The Filer prompts with this question:

```
Dup_link what file?
```

   Respond with:

```
#5:/WORKSTATIONS/SYSTEM/SYSTEM_P,#5:/SYSTEMS/$ Return
```

That concludes the required SRM software setup. Now any workstation using the BOOT disc you have created will be able to access the SRM via logical units #5 and #45. If a workstation has high performance local mass storage such as a fixed disc, that workstation's system volume will be on the local mass storage; otherwise the SRM directory #45:/WORKSTATIONS/SYSTEMnn will be the the system volume.

It is advisable to also create a private working SRM directory for each user, in addition to the SYSTEMnn directories for each workstation. Typically a user will then use unit #45 for his system volume and #5 will be prefixed to his working directory. A good way to set this up is to create a directory such as the following one in the root directory:

```
USERS
```

Then you can add subordinate directories like the following for each user:

```
USERS/TOM
USERS/DICK
USERS/HARRIET
```

SRM as the System Volume

At this point, you can make the /WORKSTATIONS/SYSTEMnn the system volume. You will first need to re-execute the TABLE program in order for unit #45: to be assigned to this directory. Press [X] at the Main Command Level, and enter this file specification:

```
/WORKSTATIONS/SYSTEMnn/TABLE.
```

Of course you will need to replace the nn with the node number of your workstation. Don't forget the period.

Now you can execute the Newsysvol command (at the Main Command Level) and specify `#45:` as the unit number. Then use the What command to verify that all of the subsystems (EDITOR, FILER, etc.) were found in the /WORKSTATIONS/SYSTEMnn directory. Changing the system volume will allow you to access the SRM copies of these subsystems by pressing keys such as [E] for Editor, and so forth.

Note

You should not prefix the working directory of unit #45: away from this directory.

Adding Modules to INITLIB

Now we will add modules DATA_COMM and SRM (on the CONFIG: disc) to INITLIB (on the BOOT: disc); actually, you will make a new INITLIB on the SRM that includes the drivers required for the SRM.

1. At the Main Command Level, press [L] to load the Librarian (note that the Librarian should be loaded from the SRM).
2. When you see the Librarian's prompt line at the top of the CRT, press [O] to specify the name of the (Output) file the Librarian will be creating.
3. Enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITNEW
   ```

4. Press [I] so you can specify an Input file, then enter:

   ```
   #5:/WORKSTATIONS/SYSTEM/INITLIB.
   ```

   Be sure to type the period after the word INITLIB in this command (to suppress the otherwise automatic .CODE suffix). The Librarian will respond by showing INITLIB as the name of the input file.
5. Near the bottom of the CRT you will see a line which says:

   ```
   M input Module: KERNEL
   ```

   Press [T] to transfer this module to the output file. After a few moments, the name of a new module (KBD) will appear. Each time a new module name appears, press T to move it to the output file. You should continue copying modules until the name LAST appears; **Don't copy the module LAST yet.**
6. Now you must get the required SRM drivers and include them in the Output file. First close the Input file by typing an [I] and then entering a null response.
7. Press [I] for an Input file and enter this file specification:

   ```
   #5:/WORKSTATIONS/SYSTEM/DATA_COMM.
   ```

   Don't forget the period after the name.
8. When the module name DATA_COMM shows up near the bottom of the screen, press [A] which tells the Librarian to transfer all the modules in the file.
9. Then use the I command again to pick up the SRM input file, again being sure to type the period after the file name:

   ```
   #5:/WORKSTATIONS/SYSTEM/SRM.
   ```

10. Again transfer All by typing [A].
11. Enter an [I] (Input file) command with null response. This closes the SRM file.
12. Press [I] for Input and enter the file specification of the original INITLIB file:

    ```
    #5:/WORKSTATIONS/SYSTEM/INITLIB,
    ```

13. When module KERNEL shows up near the bottom of the screen, select module LAST instead by pressing [M] for module and enter:

    ```
    LAST
    ```

    Then transfer it by typing [T].
14. You now have all the modules in your new library. "Keep" it by typing [K]. Then quit the Librarian by typing [Q].

Replacing INITLIB

Where you place the new version of INITLIB depends on which Boot ROM is in your machine.

- If you have Boot ROM 3.0 or later (but not 3.0L), then you will probably want to leave it in the /WORKSTATIONS/SYSTEM directory; it will be found there automatically when you boot from the SRM system.
- If you have an earlier Boot ROM or Boot ROM 3.0L, then you will need to replace the INITLIB on the BOOT: disc with the new INITLIB; this is required because these Boot ROMs cannot boot directly from SRM – they must use the BOOT: disc.

With Boot ROMs 3.0 and Later

1. Use the Filer's Change command to re-name the existing INITLIB (in /WORKSTATIONS/SYSTEM) to something like OLDINITLIB.
2. Use the Change command again to re-name the INITNEW file to INITLIB.
3. Re-boot your workstation to verify that the new INITLIB file works correctly.
4. Use the Filer's Dup-link command to link the new INITLIB to all /WORKSTATIONS/SYSTEMnn directories for the workstations that will be booting from the SRM. (You can alternately make custom INITLIB files for each workstation, if desired.)

With Earlier Boot ROMs

1. Press F to invoke the Filer.
2. Put in the spare copy of the BOOT: disc (*not* the original) into a drive. Press R for the Remove command. The computer responds with this prompt:

```
Remove what file?
```

3. Answer:

```
BOOT:INITLIB
```

   Note that there is no period after the file name this time.

4. Press K (Krunch) to pack all the remaining files on the disc to make the maximum amount of room for the new INITLIB. The Filer answers:

```
Crunch what directory?
```

5. Answer:

```
BOOT:
```

   Don't fail to type the colon after the volume name!

   The Filer will then say:

```
Crunch directory BOOT ? (Y/N)
```

6. Answer Y. The computer then prompts:

```
Crunch of directory BOOT in progress
DO NOT DISTURB!!
```

Note

If you interfere with the disc before the Crunch operation completes, you will ruin the data on the disc. You will certainly have to recopy it from the original BOOT: and you may have to re-initialize it.

After the Krunch is complete, the filer prompts:

```
Crunch completed
```

7. Now when the Crunch is finished, you can Filecopy the INITNEW library file onto the new BOOT: disc. At the same time, you can re-name it INITLIB.

   Insert disc NEWLIB and press F for the Filecopy command.

```
Filecopy what file?
```

   Answer:

```
#5/WORKSTATIONS/SYSTEM/INITNEW.CODE,BOOT:INITLIB
```

   When the Filecopy finishes, you have a BOOT:INITLIB disc which contains the SRM drivers.

8. Verify that the new INITLIB works by re-booting your system.

Each Pascal workstation in the system with earlier (or 3.0L) Boot ROMs must boot using an INITLIB which has the SRM driver software installed. You may wish to make copies of the disc you've just created for each workstation. The disc can be copied using this Filer command sequence: F *3,*3. (You can alternately make custom INITLIB files for each workstation, if you want.)

Multi-Disc SRM

When an SRM system has more that one hard disc, you will need to modify, recompile, and execute the CTABLE program to allow access to these discs. This section describes how to perform this type of configuration change.

When more than one hard disc is installed on the SRM system, each disc must have a /WORKSTATIONS directory. If the directory is write-protected, then a /WORKSTATIONS/TEMP_FILES directory must be created. You may also wish to create another /SYSTEMS directory. Boot ROM 3.0 and later versions will search for bootable systems on each disc containing a /SYSTEMS directory.

CTABLE Modifications

Near the end of the CTABLE program, just above the manual `templates` section, a small section of code assigns Unit Table entries for the SRM.

```
with SRM_dav do
  begin
    {  tea_srm( 46, sc, ba, du);  {free}
    tea_srm( 45, sc, ba, du);  {for possible use as the system unit}
  end; {with}
```

The first `tea` entry provides a template for assigning unit #46: to the second hard disc connected to the SRM. You should change the `du` parameter to `du+1` to specify the second disc.

Just below the manual "templates" section of the CTABLE program is another section pertaining to units for the SRM.

```
{ prefix the primary and secondary SRM unit entries }

if not unit_prefix_successful('#5:/') then {do nothing};
  {tries to set up uvid for possible default unit assignment below}

{ if not unit_prefix_successful('#46:/?') then zap_assigned_unit(46);  {free}

if not unit_prefix_successful('#45:'+srmsysprefix+srmnode(unitable^[45],sc)) then
  if not unit_prefix_successful('#45:'+srmsysprefix) then
    zap_assigned_unit(45);
```

If you remove the leading comment delimiter ({) from the #46: entry and remove the question mark from the literal `'#46/?'`, then Pascal will be able to recognize the second hard disc connected to the SRM.

If you wish to have a Unit Table entry for a particular directory path name, you can include the path name in the specification. For example:

```
if not unit_prefix_successful('#46:/USER/AL') then zap_assigned_unit(46);
```

If you make this modification be sure to activate its accompanying `tea_srm` procedure by removing the curly brace.

```
tea_srm( 46, sc, ba, du); {free}
```

With this modification, the system will boot with unit #46 assigned to the directory "/USER/AL" on the first SRM disc.

After all modifications have been made, you can compile CTABLE. Remember that you need to enable the `$search 'CONFIG:INTERFACE'$` Compiler option at the beginning of the program and make the INTERFACE library accessible at compile time. You will probably also want to link the resultant TABLE object file to itself with the Librarian to conserve disc space. See the procedures in the preceding section called Modifying the TABLE Program for explicit details.

Non-Disc Mass Storage

Chapter 19

Introduction

Pascal 3.0 and later versions support several types of "non-disc" mass storage:

- Internal memory (RAM)
- HP 98259A Magnetic Bubble Memory cards
- HP 98255A EPROM (Erasable Programmable Read-Only Memory) cards
- DC600 Tape Drives (found in CS80-type disc drive units)

The Bubble and EPROM cards and tape drives provide non-volatile mass storage of programs and data; internal memory is volatile. All of them can be accessed through the File System. However, Pascal will not recognize either Bubble or EPROM cards until a few modifications are made.

This chapter describes configuring and accessing Bubbles, EPROMs, and tapes. Using internal memory for mass storage is covered in the *Pascal User's Guide* and in the description of the Memvol command in the "Main Command Level" chapter of this manual.

Summary of Configuration Modifications

In order for the File System to recognize either the Bubble Card or the EPROM card, you need to make the following configuration changes:

- Add the appropriate driver-module to INITLIB
- Modify the TABLE auto-configuration program. The source program (CTABLE) already contains the necessary templates; you only need to make a few simple changes to enable them.

Tape drives will be recognized without changes to INITLIB or the TABLE auto-configuration program.

Mass Storage Comparison

The operating characteristics for various mass storage devices are compared in the following table.

Characteristics	Storage Device				
	Mini-Floppy Discs	**Bubble Cards**	**EPROM Cards**	**Memory Volumes**	**DC600 Tapes**
Storage Capacity	270 336	131 072	131 072[1] or 262 144	variable[2]	16 000 000 or 67 000 000
Relative Access Speed	moderate	slow	fast	fast	slow
Read/Write Capability	yes	yes	no[3]	yes	yes
Usable as a Boot Device	yes	yes[4]	yes[5]	no	no
Removable Media	yes	no	no	no	yes
Multiple Volumes	no	no	yes[6]	no	yes
Data Integrity	poor	good	good	moderate[7]	good
Relative Cost	low	high	moderate	moderate	low

1 Size depends on EPROM device type. Sixteen 2764-type devices provide 131 072 bytes while sixteen 27128-type devices provide 262 144 bytes.

2 Size is limited by available memory.

3 EPROMs can be read just like RAM memory but must be programmed (written) with the HP 98253 EPROM Programmer Card.

4 The CTABLE program must be modified to allow this boot device to be the default system volume.

5 This device can be allowed as a boot device; however, there are several restrictions that apply. See the discussion of Booting from EDISCS for further information.

6 Multiple volumes can be programmed into one EPROM Card, on 16 Kbyte boundaries.

7 RAM memory reliability is dependent on power-source stability.

Using Bubble Cards

This section provides all of the information you will need to configure and access Bubble memory cards from the File System.

Power Constraints

Due to the amount of power consumed by a Bubble card when data is being transferred, no more than two Bubble cards can operate at the same time without exceeding the capacity of the power supply in the existing Series 200/300 Computers. It is further recommended that only one Bubble card be operating at the same time as any other "high-power" card (such as the HP 98620 DMA card).

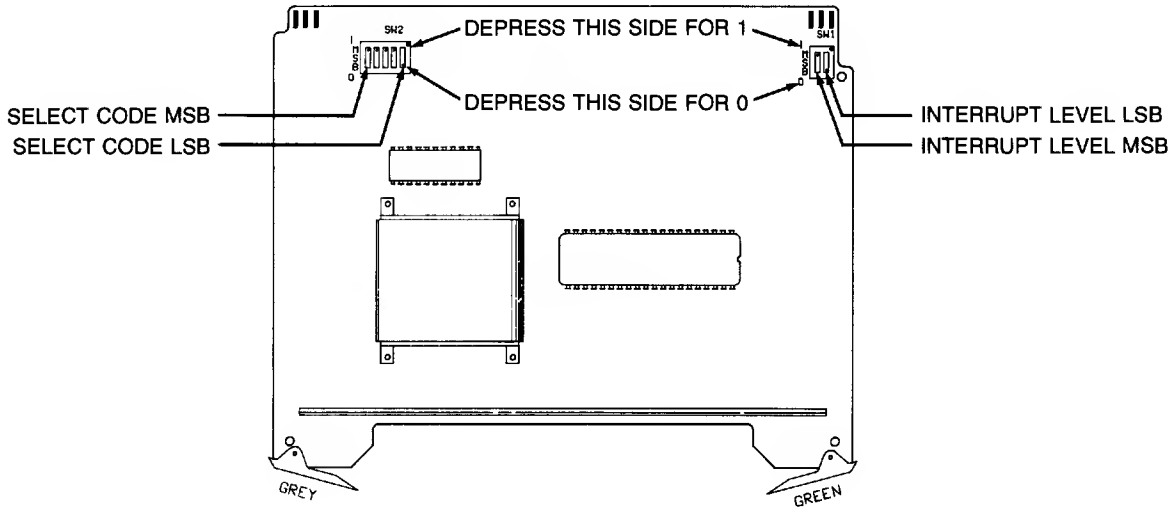
Bubble Card Configuration

If you have not already installed the Bubble card, see its installation note for complete details. Some of the installation information is repeated below for convenience.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

The Bubble card has two banks of switches. The large switch bank sets the select code while the small one controls the interrupt priority.



Bubble Card Switch Locator

Select Code

The Bubble card's select code is preset at the factory to select code 30. If this select code conflicts with any another interface present in the system, change it to some *unused* value from 8 through 31. Note the select code setting; it will be needed for the changes to the TABLE program.

Note

If you change the select code of the Bubble card from its factory default setting, you must also change the CTABLE program accordingly.

Select Code Switch Settings

Switch Settings MSB 43210 LSB	Select Code	Switch Settings MSB 43210 LSB	Select Code
01000	8	10100	20
01001	9	10101	21
01010	10	10110	22
01011	11	10111	23
01100	12	11000	24
01101	13	11001	25
01110	14	11010	26
01111	15	11011	27
10000	16	11100	28
10001	17	11101	29
10010	18	**11110**	**30**
10011	19	11111	31

Interrupt Priority

The interrupt priority switches have been preset to level 5. Each Bubble card should be set to a *unique* interrupt priority since the Bubble card may lose data if interrupts are not serviced quickly. This is especially true if you plan to make calls directly to the driver procedure or use the overlapped I/O capability.

Interrupt Priority Switch Settings

Interrupt Level	Setting MSB	LSB
3	0	0
4	0	1
5	1	0
6	1	1

If other interfaces have been installed which use interrupt level 5, change the switches on the Bubble card to the highest unused interrupt level in the range 3 through 6.

The Bubble card should now be ready to install in the computer. With the power turned *off*, install the card in the backplane. See the installation note if you have any difficulties.

INITLIB Driver Modules

The BUBBLE module is supplied on the Pascal 3.0 CONFIG: disc. The Pascal 3.0 IODECLARATIONS module recognizes Bubble cards as CARD_TYPE = 8 (a field of the ISC_TABLE array in IODECLARATIONS). This same version also recognizes the EPROM cards.

Loading the BUBBLE Module

As with other driver modules, there are two ways to load the BUBBLE module:

- Execute it (with the Main Command Level eXecute command)
- Add it to INITLIB

Executing the module "permanently" loads the module, but must be performed every time the system is booted. Adding the module to INITLIB eliminates having to load the module each time and allows the Bubble card to be a candidate for use as the system volume.

Adding BUBBLE to INITLIB

If you have two disc drives, the creation of the new INITLIB is relatively simple. If you only have one disc drive, you will need to create a memory volume large enough to hold the new library (about 200 Kbytes).

To create a memory volume, press [M] from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 400) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the "Main Command Level" chapter of this manual.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the disc.

   (It is a very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are, no matter what happens to the new disc.)
2. Invoke the Librarian. This is done by pressing [L] from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.
3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)

4. Now use the Librarian to create the new INITLIB.

   a. Press [I] and then enter the file specification by typing `#3:INITLIB.` and [Return] or [ENTER]. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

      When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press [F] to list all of the modules in INITLIB.

      The BUBBLE module can be inserted anywhere after the IODECLARATIONS module but *before* the module named LAST. (LAST must be the last module in INITLIB.)

   b. Press [O] and enter `#4:BUBLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

      (This disc must *not* be removed until you have finished creating the new BUBLIB file.) If you are using a memory volume, use the unit number of the memory volume.

   c. Press [E] to enter the Edit mode. You should now see this prompt (in the middle of the screen):

```
F  First module: KERNEL
U  Until module: (end of file)
```

   d. Press [U] and enter `LAST` as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing [C].

   e. When the preceding transfer is complete, press [A] to append a module to the BUBLIB Output file. The Librarian prompts with `Input file:`. Put the CONFIG: disc, or whichever disc now contains the BUBBLE module, in Unit #3 (*not* #4, which must not be removed). Enter this file specification: `#3:BUBBLE.`.

   f. The Librarian now prompts with `Enter list of modules or = for all.` Enter `=`. After the BUBBLE module has been transferred to the BUBLIB library, the Librarian prompts with `Append done, <space> to continue.` If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the CONFIG: disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

      (If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message:

```
cannot open '#3:INITLIB', ioresult = 10.
```

      In such case, don't worry. Remove the CONFIG: disc and insert the BOOT: disc, then press [I] and enter `#3:INITLIB.` as the Input file. Press [E] to return to Edit mode, and return to where you were previously by pressing [F] and entering `LAST` as the First module. Proceed with step g below.)

   g. Press [T] to transfer module LAST to the BUBLIB file (if you got the error described in the preceding step, press [C] instead of [T]). Then press [S] to stop editing and [K] to keep the file.

   h. You should now verify that the BUBBLE module was indeed copied to the Output file. Press [I] and enter `#4:BUBLIB.` as the new Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press [F] to get a File Directory listing.

   i. If all modules are present, then press [Q] to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named BUBLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy BUBLIB from the memory volume to the new disc, changing the file name to INITLIB in the process.
6. Re-boot the computer, which installs the new INITLIB containing the BUBBLE module.

Once the BUBBLE module has been installed, the Bubble card can be accessed by procedure calls. (The procedure calls will be discussed later.) To make the Bubble card available to the file system as a mass storage unit, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The Pascal 3.0 CTABLE program, supplied on the CONFIG: disc, contains a "template" for the Bubble card. You can either use the Editor's Find command to search through all occurrences of `BUBBLE` until you find the template, or Jump to the end of the program and scroll up until you see the BUBBLE template shown below.

```
$if false$ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

Change `$if false$` to `$if true$`.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
$end$
```

This is the only modification that must be made to CTABLE for the system to recognize the Bubble card. It assigns unit number 42 to the Bubble card. If you are already using unit number 42, change the unit number to one that you are not using.

If you have more than one Bubble card or wish to have the Bubble card as a possible system volume, you should consider the following modifications.

Multiple Bubble Cards

If you install more than one Bubble card, a separate "tea" procedure must be executed for each card. An example is shown below.

```
$if true $ { BUBBLE memory }
    {watch for conflicting uses of unit 42}
    {BUBBLE_DAV.SC default is 30 but may have been changed to boot SC}
    tea_BUBBLE(42,primary_dam,BUBBLE_dav.SC);
    tea_BUBBLE(20,primary_dam,28);
    tea_BUBBLE(21,primary_dam,29);
    { tea_BUBBLE(3,primary_dam,30); {This would override #3}
$end$
```

For each tea_BUBBLE procedure called, you should specify an unused unit table entry, the type of directory access method (the LIF DAM is recommended), and the select code (switch setting) of the Bubble card. Since these templates override the auto-configuration, the last entry in the above example would have overridden the device otherwise assigned to unit #3.

You can use the Filer's Volumes command to determine what units are being used. However, remember that some devices have a second unit number assigned for an alternate DAM that the Volumes command does not display. For example, the device which has the unit #3 (LIF DAM) entry also has the unit #43 (UCSD DAM) entry.

Bubbles as the System Volume

The Pascal 3.0 TABLE program *already* contains code to support BUBBLE memory as a default system volume. This support is declared in the `{system unit auto-search declarations}` constants near the end of the "options" module. This constant tells TABLE to search through 7 possible system volumes. Note that unit number 42 (the default for the Bubble card) is included in the list. If you used a unit number other than 42 for the Bubble card, be sure to change the unit number in the search list above.

Compiling CTABLE

After all modifications have been made to the CTABLE program, the program must be compiled. If you do not know how to compile a Pascal program, see the Compiler chapter for details.

The resulting code file will be linked and stored as TABLE on the new BOOT: disc. Since CTABLE imports several operating system modules, you will need to make the CONFIG:INTERFACE file accessible to the compiler (this file contains the interface text for the operating system modules). To do so, you can either "un-comment" the following compiler option (near the beginning of CTABLE.TEXT):

```
$Search 'CONFIG:INTERFACE'$
```

or add the `CONFIG:INTERFACE` file to the current System Library file. The linking procedure is described next.

Linking CTABLE

Once CTABLE.TEXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of CTABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled and resides in unit #3 as CTABLE-.CODE. Since the linked version of CTABLE is usually less than 15K bytes, it will be put on the same disc that contains the original CTABLE.CODE file (probably CONFIG:) and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (Output) file directly onto the new BOOT: disc.

1. Press [L] to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press [I] and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press [H] to specify a new Header size; enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press [O] and enter `#3:TABLE.` as the Output file name. The trailing period will suppress the `.CODE` suffix.

5. Perform the actual linking.Syntax:
   a. [L] – to Link. This will update the display.
   b. [D] – to toggle the Define source (export text) to NO.
   c. [A] – to transfer All modules.
   d. [L] – to finish Linking.
   e. [K] – to Keep the Output file.
   f. [Q] – to Quit the Librarian.

6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did **not** include the BUBBLE module in the INITLIB, the File System will not recognize the Bubble card until you execute the BUBBLE module.

7. Re-boot the system using the new BOOT: disc. Pascal will now recognize the Bubble card.

Bubble Cards in the File System

After the BUBBLE module is installed and an appropriate TABLE program is executed, the Bubble card appears to the File System as a non-removable blocked device (mass storage volume). Any of the local mass storage directory access methods (DAMs) may be used, however the LIF DAM is recommended to allow use of the unit as a boot device.

Executing the Filer's Volumes command will now show that a unit number has been assigned to the Bubble card. For example:

```
Volumes on line:
  1    CONSOLE:
  2    SYSTERM:
  3 #  ACCESS:
  4 *  SYSVOL:
  6    PRINTER:
 42 #  VBUB:
Prefix is - ACCESS:
```

Unlike discs, Bubble memory units are initialized with the LIF DAM before being shipped. This means there is already a directory on the Bubble media. Use the Filer's List command to see the directory. For example, from the Main Command Level, press [F] to access the Filer, and then use the List directory command by pressing [L]. Specify #42 (or whatever unit number is assigned to Bubbles). Here is a typical display:

```
VBUB:                Directory type= LIF level 1
created 14-Apr-84 16.21.25 block size=256
Storage order
...file name....     # blks     # bytes   last chng

FILES shown=0 allocated=0 unallocated=8
BLOCKS (256 bytes) used=0 unused=509 largest space=509
```

You can now use the Bubble card as you would any other LIF mass storage volume. It can be zeroed (all files removed) by the Filer's Zero command and it can be initialized by the MEDIAINIT program supplied with the system.

The Bubble Memory cards have access and timing characteristics similar to the HP 9826/36 Computers' internal mini-floppy mass storage drives.

Your Bubble card should provide years of reliable, error-free operation. If you ever have cause to suspect the reliability of the Bubble card, make a back-up copy and and then try re-initializing the card before contacting your Sales and Service Office.

Error Correction

The Bubble Memory unit shipped to you has automatic error correction enabled. If some other memory unit (the hardware package containing the magnetic bubbles) is ever installed in the Bubble card, it should first be initialized by MEDIAINIT to ensure that automatic error correction is enabled.

The Bubble Device

The Bubble Memory unit installed in the Bubble card is a very stable non-volatile storage system. It is not easily damaged by external magnetic fields or mechanical abuse. It is, however, *strongly* recommended that the memory unit not be removed from the card. Removal of the memory unit from the card may damage the "boot loop" or the "seed bubbles".

The boot loop of a Bubble card is equivalent to the spared tracks record of a disc. If the boot loop is damaged, the memory will not function properly. A damaged boot loop may appear as permanent read/write errors, or more likely it will be detected by the TM (Transfer Method) when a UNITCLEAR operation is performed and reported as bad hardware. UNITCLEAR is performed on all units by the TABLE program and by a CLEAR I/O operation initiated from the keyboard (using the Stop key).

The memory of a Bubble unit is organized in tracks similar to a disc. Since a bit of information is indicated by the presence or absence of a bubble, information is written to a track by destroying or creating magnetic bubbles.

A magnetic bubble is created by spliting a seed bubble. If the bubble unit is removed or improperly installed a seed bubble may be destroyed or lost. This condition will appear as permanent read/write errors. If you suspect your bubble unit has been damaged in this way, contact your HP Sales and Service Office.

Initialization

The MEDIAINIT program on the ACCESS: disc is capable of initializing a BUBBLE device. The initialization process writes blanks to every location on the media, then writes a default directory to the unit. The only time MEDIAINIT should have to be used is when a Bubble Memory device not supplied by HP is placed on the card.

The Filer's Zero command can be used to remove all the files in the Bubble card. The procedure is similar to the Zero operation of discs. You can change the volume name and the number of directory entries but you should accept the default value for the size of the media.

If you do choose to initialize the Bubble card, execute the MEDIAINIT program and supply the appropriate unit number. Use the default value for all questions.

Interrupts and Overlapped I/O

Bubble devices require immediate interrupt service of relatively short duration. Since the Pascal Workstation File System performs *only* serial I/O, the problem of interrupt priority selection is reduced to ensuring that the BUBBLE module is placed in INITLIB after all other driver modules (but before module LAST). This will ensure that Bubble cards are checked before any other devices (on the same interrupt level) and therefore minimize the time required to service an interrupt.

When performing overlapped Bubble-card-to-Bubble-card transfers, best results are achieved when the destination priority is lower than, or the same as, the source priority. A priority configuration other than this will result in even poorer performance than if non-overlapped I/O is used because the two devices interfere with each other and cause several re-tries per transfer. This is not a problem on machines equipped with cache-memory hardware.

Using EPROM Memory

This section introduces you to the programming and operating characteristics of the HP 98255 EPROM card and the HP 98253 Programmer card. With these cards and Pascal 3.0 (or later versions), you can copy files and volumes into EPROMs.

Overview

EPROMs are high-speed memory devices used for storing programs or other information. The HP 98255 EPROM card and the HP 98253 Programming card support 2764, 27128, and equivalent types of EPROMs.

The EPROM devices are not supplied with the EPROM card. You will have to purchase them separately through an electronic-supply vendor or other source. You probably will also need to purchase an ultra-violet (UV) light source to erase the EPROMs.

The storage capacity of an EPROM can be determined by the final digits of the part number. For example, a 2764-type device contains 64 Kbits (65 536 bits) while a 27128-type device contains 128 Kbits (131 072 bits). Up to 16 EPROMs can be placed on one card; this means that one card provides 131 072 bytes of storage using the 2764-type EPROMs or 262 144 bytes using 27128-type EPROMs.

The data in an EPROM can be read just like RAM memory, however, a special process is needed to program (write) the data into EPROMs. The HP 98253 Programmer card is used for this purpose. An EPROM is programmed by applying a short "burn" pulse while the data being programmed is applied to the output pins. The timing and control of this operation is handled by the Programmer card. Once the EPROMs have been programmed, the Programmer card is no longer needed in the system and can be removed. (With the power turned-off of course!)

An EPROM can be erased (all bits set to "1") by exposing it to a high level of ultra-violet light. Once erased, the EPROMs can be reprogrammed with new data. Check the EPROM manufacturer's specifications for details on the type of UV light source needed and the recommended exposure time.

Configuration Changes Required

There are two changes you need to make to the "standard" configuration in order to use EPROMs:

- Add module(s) to INITLIB.
- Modify the TABLE source program (CTABLE.TEXT)

You may also need to set switches on the cards and install EPROM devices.

INITLIB Driver Modules

In addition to supporting the operations of the HP 98255 EPROM card and the HP 98253 Programmer card, Pascal 3.0 supports the use of EPROMs as a mass storage volume. Transferring a volume into EPROMs would create what could be called an "Eprom-DISC" or "EDISC".

The support modules include:

- The EPROMS module is included on the CONFIG: disc. The module may be installed by either executing it or by using the Librarian to include it in INITLIB.
- The EDRIVER module, also included on the CONFIG: disc, provides *read/write* capability for performing various operations with an EPROM and Programmer card pair. The EDRIVER module can be "P-loaded" or linked to an application program when read/write capability is needed.
- The EPROM Transfer Utility (ETU.CODE) included on the ACCESS: disc allows mass storage volumes to be transferred to EPROMs. When environmental conditions limit the reliability of floppy discs, or when it is desirable to have quick access to commonly used programs or data, a copy of a mass storage volume can be transferred to EPROMs. Transferring a volume to EPROMs creates an "EDISC".

  ETU can also be used to transfer DATA, ASC, and TEXT files to EPROMs. This capability allows arbitrary bit-patterns to be transferred to EPROMs.
- The CTABLE.TEXT file on the CONFIG: disc contains a "template" section to assign unit numbers to EDISCs.
- The Pascal 3.0 IODECLARATIONS module recognizes a Programmer card as CARD_TYPE = 9 (a field of the ISC_TABLE array). This same version also recognizes Bubble memory cards.

You do not have to load any modules before using the ETU program since it already has the necessary drivers included in its code. When you are finished programming the EPROMs, the EPROM module should be added to INITLIB to provide access to EPROMs. This process is described later in this chapter.

Programmer Card Installation

If you have not already installed the Programmer card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

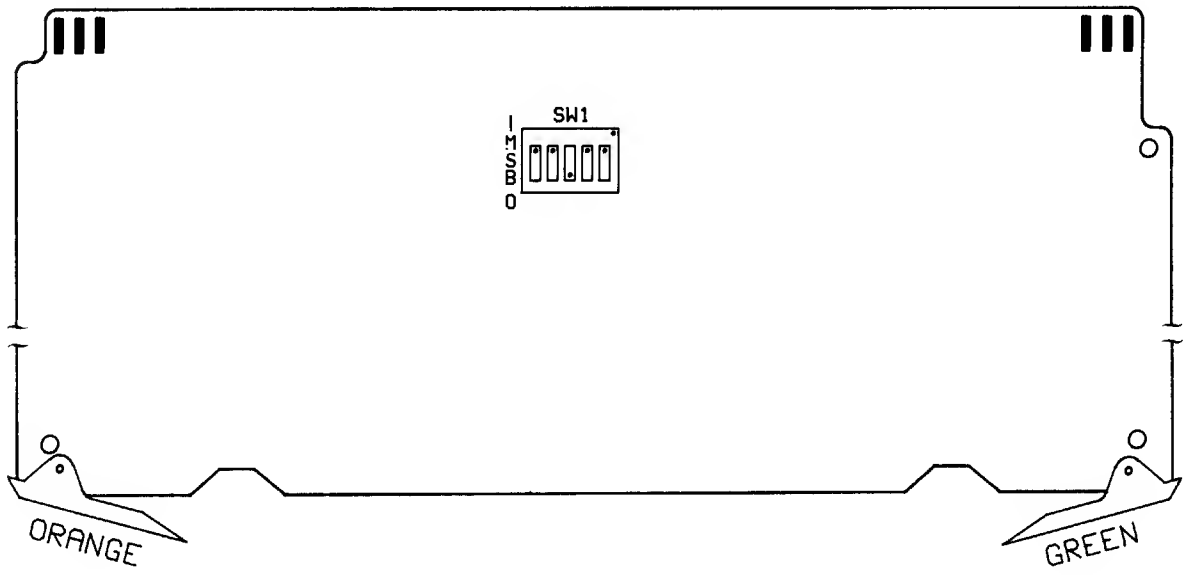
The purpose of the HP 98253 Programmer card is to program (write) information into the EPROMs on the HP 98255 EPROM card. Once the information has been programmed, the Programmer card can be removed from the computer's backplane.

CAUTION

ALWAYS TURN THE COMPUTER OFF BEFORE INSTALLING OR REMOVING INTERFACES.

Perform the following steps to install the Programmer card:

1. Check the select code switch on the Programmer card. The HP 98253 Programmer card's select code has been preset to 27 at the factory. If this conflicts with any other I/O card in the system then change it to an unused select code. If more than one Programmer card is installed, set each card to a unique select code.



Programmer Card Switch Location

Select Code Switch Settings

Switch Settings MSB 43210 LSB	Select Code	Switch Settings MSB 43210 LSB	Select Code
01000	8	10100	20
01001	9	10101	21
01010	10	10110	22
01011	11	10111	23
01100	12	11000	24
01101	13	11001	25
01110	14	11010	26
01111	15	**11011**	**27**
10000	16	11100	28
10001	17	11101	29
10010	18	11110	30
10011	19	11111	31

2. With the computer power turned *off*, install the Programmer card in the computer's backplane. The Programmer card's ribbon cable will be connected to an EPROM card later.

When more than one Programmer card or EPROM card is installed at the same time, the ribbon cable can be connected to different EPROM cards without turning off system power. Be sure that no read or write operation is taking place when the cable is exchanged.

CAUTION

THE PROGRAMMER CARD'S CABLE MUST **NOT** BE REMOVED OR CONNECTED WHEN THE EPROM CARD IS IN USE.

A small light-emitting diode (LED) on the Programmer card indicates when system power is on. (It does **not** indicate when the card is in use.)

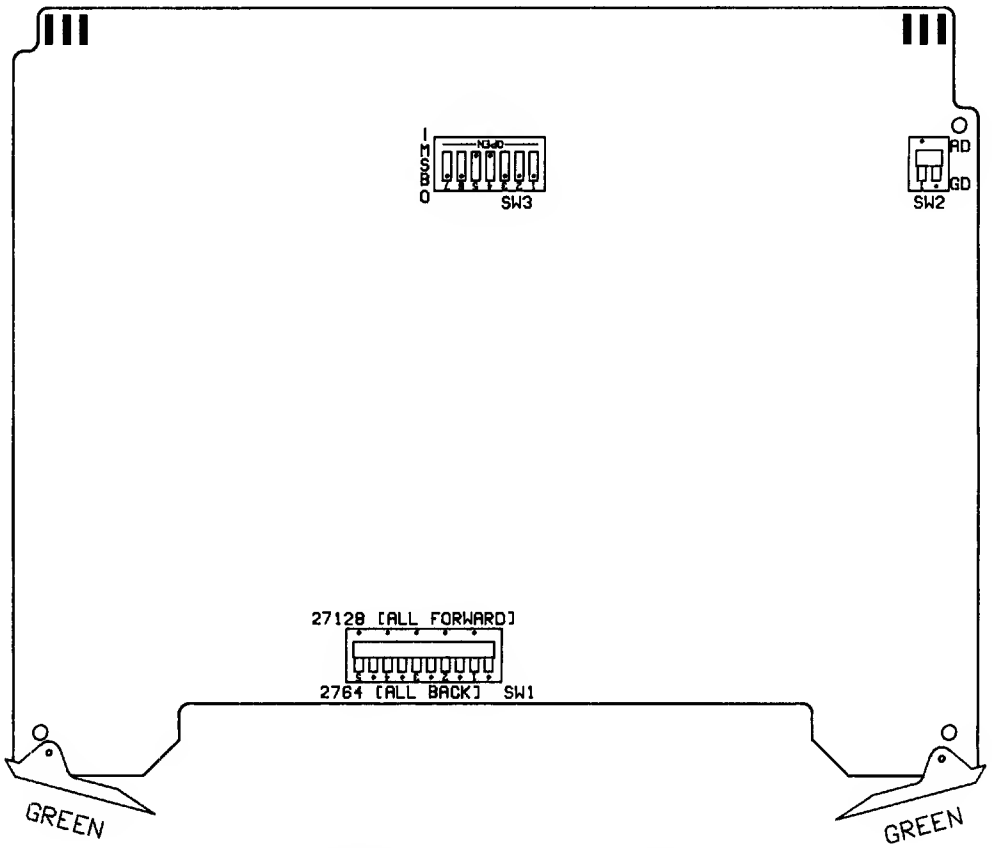
EPROM Card Installation

If you have not already installed the EPROM card, see its installation manual for complete details. Some of the installation information is repeated here for convenience.

Before installing an EPROM card in the computer's backplane, you need to check and set the card switches. There are three sets of switched on the card.

- EPROM-type switch (SW1)
- Address-response switch (SW2)
- Address switch (SW3)

The position of these switches is shown in the following illustration:



EPROM Card Switch Locations

The largest switch is the "EPROM-type" switch. It tells the card's hardware what capacity of EPROM to expect. All segments of this switch are "ganged" together to configure all 16 sockets for either 2764-type or 27128-type EPROMs. You cannot mix two different types of EPROMs on one card, but you do not need to completely fill all 16 sockets with EPROMs. If you only partially fill the card, use pairs of EPROMs (upper and lower byte socket-pairs) and fill the lowest numbered sockets first.

The smallest switch on the card is the "DTACK" switch and it controls the card's response when it is addressed (i.e. whether it should respond like ROM or RAM memory). This switch, which can be set for AD (Automatic DTACK) or GD (Generate DTACK), must be set to AD for the EPROM card to appear in the computer's ROM memory space.

Note

The modules provided with Pascal 3.0 only support EPROM cards which are addressed in the ROM address space. Set the "DTACK" switch to AD (Auto-DTACK).

The third switch determines the base memory address of the card. Special care must be taken to ensure that the address space selected does not overlap another EPROM card or ROM card. The EPROMs on the card are "memory mapped" (in pairs) by ascending socket number. For example, byte 0 is the first location in socket 0U, byte 1 is the first location in socket 0L, byte 2 is the second location in socket 0U, byte 3 is the second location in socket 0L, and so on.

Note

If you have a ROM-based language system, do **not** set the EPROM card's switches to the same address space used by the ROM Language. For instance, the ROM version of the HPL Language System is addressed at $10 0000 and extends up to $12 0000. The ROM 1.0 version of the BASIC Language is addressed at $2 0000 and extends to $2 4000, while the ROM 2.x versions begin at $8 0000 and vary in size.

To see where these ROM-based systems reside, you can check for the presence of "ROM headers" (contents $F0FF) which are located on 16 Kbyte boundaries, beginning at $2 0000 and extending through the Auto-DTACK range of addresses. Auto-DTACK extends to $20 0000 for cache-memory processor boards (i.e., machines with "U" suffix such as the HP 9836U) and $40 0000 for non-cache-memory processor boards (such as the HP 9836A).

Although the switches can be set to make the EPROM card appear almost anywhere in the computer's address space, the following table shows the recommended settings. When the smaller capacity EPROMs are used, multiple cards can be addressed 128 Kbytes apart; cards filled with the larger capacity devices must be addressed 256 Kbytes apart.

Address Switch Settings

Switch Settings MSB LSB	EPROM Card's Base Address for Programming: Hex Start Address	Decimal Address (2764-type devices)	Decimal Address (27128-type devices)
0 0 0 0 0 0 0 1	$2 0000	131 072	
0 0 0 0 0 0 1 0	$4 0000	262 144	262 144
0 0 0 0 0 0 1 1	$6 0000	393 216	
0 0 0 0 0 1 0 0	$8 0000	524 288	524 288
0 0 0 0 0 1 0 1	$A 0000	655 360	
0 0 0 0 0 1 1 0	$C 0000	786 432	786 432
0 0 0 0 0 1 1 1	$E 0000	917 504	
0 0 0 0 1 0 0 0	$10 0000	1 048 576	1 048 576
0 0 0 0 1 0 0 1	$12 0000	1 179 648	
0 0 0 0 1 0 1 0	$14 0000	1 310 720	1 310 720
0 0 0 0 1 0 1 1	$16 0000	1 441 792	
0 0 0 0 1 1 0 0	$18 0000	1 572 864	1 572 864
0 0 0 0 1 1 0 1	$1A 0000	1 703 936	
0 0 0 0 1 1 1 0	$1C 0000	1 835 008	1 835 008
0 0 0 0 1 1 1 1	$1E 0000	1 966 080	
0 0 0 1 0 0 0 0	$20 0000	2 097 152	2 097 152
0 0 0 1 0 0 0 1	$22 0000	2 228 224	
0 0 0 1 0 0 1 0	$24 0000	2 359 296	2 359 296
0 0 0 1 0 0 1 1	$26 0000	2 490 368	
0 0 0 1 0 1 0 0	$28 0000	2 621 440	2 621 440
0 0 0 1 0 1 0 1	$2A 0000	2 752 512	
0 0 0 1 0 1 1 0	$2C 0000	2 883 584	2 883 584
0 0 0 1 0 1 1 1	$2E 0000	3 014 656	
0 0 0 1 1 0 0 0	$30 0000	3 145 728	3 145 728
0 0 0 1 1 0 0 1	$32 0000	3 276 800	
0 0 0 1 1 0 1 0	$34 0000	3 407 872	3 407 872
0 0 0 1 1 0 1 1	$36 0000	3 538 944	
0 0 0 1 1 1 0 0	$38 0000	3 670 016	3 670 016
0 0 0 1 1 1 0 1	$3A 0000	3 801 088	
0 0 0 1 1 1 1 0	$3C 0000	3 932 160	3 932 160
0 0 0 1 1 1 1 1	$3E 0000	4 063 232	

Once the EPROM card's switches have been set, install the EPROM devices on the HP 98255 EPROM card. Be very careful when installing the EPROMs on the card, since the pins are easily bent. Both the EPROMs and the sockets have notches to indicate the proper orientation. See the installation manual for details.

With the power switched *off*, install the EPROM card in the computer's backplane.

Multiple EPROM Cards

If more than one blank EPROM card is installed in the computer's backplane at the same time, be sure each EPROM card is addressed to different memory locations. The lowest addressed card should be programmed first. Blank EPROM cards can not be detected by the Pascal system unless they are connected to the Programmer card.

Cable Connections

When you have finished installing the Programmer and EPROM cards, you can connect the ribbon cable from the Programmer card to the desired EPROM card. The cable connection defines and establishes the "card-pair" for programming operations.

The Programming Utility

The ETU program supplied with Pascal 3.0 supports the following operations for an EPROM and Programmer card-pair:

- Display current Programmer and EPROM card information
- Check for blank space on the EPROM card
- Transfer a mass storage volume to EPROM (EDISC creation)
- Transfer DATA, TEXT, or ASC files to EPROM (user-defined patterns)

The exact action taken in a transfer operation depends on the type of file involved. All transfers are done in two passes through the data. The two passes perform the same actions except that the data is actually programmed (written) only during the second pass.

Note

All file types other than TEXT and ASC are treated as DATA files.

Transferring Volumes to EPROM

When you specify that a volume is to be transferred to EPROM, the ETU program assumes that an EDISC is to be created. The EDISC will appear to the File System as a mass storage volume not unlike a floppy disc, but with much faster access. The maximum size of the volume depends on the capacity of the EPROMs installed in the card. The largest EDISC that can be created contains 256K bytes, since EDISCs can not cross EPROM card boundaries.

Once an EDISC has been created, you should not copy the EDISC to any other mass storage volume.

Booting from EDISC

Boot ROM 3.0 and later versions can boot from an EDISC. Booting from these devices is like booting from any other mass storage media; the system is copied into RAM and executed from there rather than from the EDISC (unlike ROM-based systems which execute from ROM).

EDISC as the System Volume

Pascal can also recognize an EDISC volume as the system volume; however, since the system volume is used by the system to store all temporary files, I/O error 18 ("Device is write-protected") will be reported whenever the system attempts to write to this "write-protected" device.

AUTOSTART and other normal stream files will not work if the system volume is an EDISC. (Normally, when a file is Streamed, the file is copied to the file named STREAM on the current system volume; this is not possible with EDISCS, since they are effectively "write-protected.") You should use the AUTOKEYS file to perform autostart functions from these devices. Other stream file names must contain a `[*]` specifier which indicates that the stream-file prompt feature is disabled. See the description of the Stream command in the Overview chapter for further details of using prompts in Stream files.

EDISC Headers

When a volume is transferred to EPROMs, an EDISC "header" is first generated and programmed into the EPROMs. The Boot ROM can detect the header and make that information available to the file system. In other words, if a volume is transferred to EPROMs, special information is added that allows the Boot ROM to detect and possibly boot from the EDISC.

Boot ROM 3.0 and later versions check for EDISC headers (and other types of headers) on 16 Kbyte boundaries, starting at 128 Kbytes ($2 0000) and continuing through the Auto-DTACK range of addresses; this range extends to $20 0000 for machines with cache-memory processor boards (i.e., computers with "U" suffix such as the HP 9836U), and $40 0000 for machines without cache-memory processor boards (such as the HP 9836A). This searching operation effectively divides the address space into 16 Kbyte "blocks".

Since the Boot ROM will check for an EDISC header on every 16 Kbyte block boundary, more than one EDISC can be programmed onto a single EPROM card. There are 8 blocks (numbered 0..7) on an EPROM card using the small capacity EPROMs and 16 blocks (numbered 0..15) using the large capacity EPROMs.

To prevent the Boot ROM from accidentally interpreting the contents of a block boundary as an EDISC header, the utility program writes binary zeros (hexadecimal pattern $0000) into the boundary locations searched by the Boot ROM. The volume's data is appropriately mapped around the block boundaries. The mapping operation is completely handled by the system, but this does mean the EDISC volume will be a few bytes larger than the original volume.

The total number of bytes needed to program a volume can be computed by taking the source volume's size and adding 18 bytes for the EDISC header and 2 bytes for each 16 Kbyte boundary crossed. If the last sector of the volume is unused, the extra bytes can be truncated without loss of data.

Transferring Files to EPROM

The ETU program can be used to transfer DATA, TEXT, or ASC type files to EPROMs. If the file type is **not** TEXT or ASC, the files will be programmed into EPROMs so as to create an exact bit for bit copy. If the file type is TEXT or ASC then *only the data parts* are programmed into EPROM (not the data separators and other "overhead"; this is equivalent to reading a line from the file into a string with a READLN statement and then "burning" the contents of the string.)

Unlike volumes, individual files transferred to EPROMs without the "directory" information of a volume cannot be detected by the file system. If you write a program to access a file that was programmed into EPROMs, you will have to tell your program where to find it. Even if only one file is to be transferred to EPROMs, you might consider putting the file in a volume and transferring the volume.

Not only do you have to keep track of the location of individual files transferred to EPROM, you must be sure that the data does not accidentally appear to the Boot ROM as a "ROM header". The Boot ROM searches for a two-byte header pattern (F0FF hexadecimal) at 16K byte intervals in the Series 200 Computer's ROM space.

The header pattern is not likely to occur in TEXT or ASCII files, however, a DATA file programmed into EPROMs may contain such a bit-pattern, and if the pattern occurs on a 16 Kbyte "block" boundary, unpredictable results may occur. The ETU program does not check for this condition.

Preparing a Transfer

Before starting the utility to transfer a volume or file to EPROMs, you must decide what you want to transfer. There are some restrictions that may influence your decision.

- The total number of bytes transferred must be less than the total capacity of the EPROMs. Excess bytes will be truncated. It is unlikely that a truncated file will be very useful.
- If the "source" volume is larger than the current available space on the EPROM card, the volume will be truncated. Since LIF volumes contain all of their directory information at the beginning of the volume, you can truncate the unused sectors at the end of a volume with relative impunity.

For the purposes of this discussion, it has been decided to transfer the Pascal Editor and Filer to EPROMs. This will allow fast access to the programs without requiring as much RAM memory as is necessary to "P-load" both of them. The number of bytes required for both the Editor and Filer is less than 120K bytes so both programs will easily fit on the EPROM card even if the smaller capacity EPROMs are installed.

Note that programs are **not** usually executed in EPROMs; rather, a copy of the program is made in RAM memory and then the copy is executed. When you quit the program, and the copy is no longer needed, the RAM memory used for the copy is free to be used by other programs. This has advantages over a program that is "P-loaded" since a "P-loaded" program remains in RAM memory until the next boot operation.

The volume containing the programs to be transferred to EPROMs should not be any larger than necessary since the entire volume will be transferred, including any unused sectors. Once the volume has been transferred to EPROMs, there is no way to go back and fill the unused sectors in the volume. Therefore, for our example, the best approach will be to create a memory volume just large enough to hold both the Editor and Filer. This will be the volume that is transferred to EPROMs.

Creating a Memory Volume

A memory volume needs two (2) "system" sectors, one (1) sector for directory information, and enough sectors to hold the files. The size of the Filer is about 228 sectors (58368 bytes), and the size of the Editor is about 232 sectors (59392 bytes). Thus, in our example, we need 3 + 1 + 460 sectors, or a total of 464 sectors.

The Memvol command "thinks" in 512-byte blocks not in 256-byte sectors. Therefore, to create the correct size memory volume, we need an even number of sectors (round-up). The total number of blocks is then 460/2 or 230 blocks.

From Pascal's Main Command Level, press [M] to create a memory volume. Answer 230 to the "Number of Blocks" question, and answer 8 to the "Number of Directory Entries" question.

When you have created the memory volume, Filecopy the Editor and Filer from the ACCESS: disc. Then use the Filer to Change the volume name from RAM: to ESYS: (the volume name will also be transferred to EPROMs).

The "Empty sockets" command of the transfer utility can be used to protect EPROMs from being programmed if there are a large number of unused sectors in the volume being transferred to EPROMs. The ETU program will then detect that the volume being transferred is larger than the available space and allow you to truncate the unused bytes. Be sure that it does not truncate part of a file!

The EPROM Transfer Utility

The EPROM Transfer Utility program (ETU.CODE) is included on the ACCESS: disc. This utility provides a convenient method of programming the EPROMs on a HP 98255 card. Either single files or entire volumes can be transferred to EPROMs with this utility.

If you haven't already executed ETU.CODE, do so now. When the utility is executed, it automatically searches for the Programmer card connected to an EPROM card. If a card is missing or incorrectly installed, you will get one of the following messages.

```
*** NO PROGRAMMER CARD IN SYSTEM ***

NO EPROM CARD ATTACHED TO PROGRAMMER CARD
```

If the system does not recognize the Programmer card, turn power off and check the select code switch settings. You should check that each switch segment is toggled correctly and that no other interface card is set to the same select code.

When the Programmer and EPROM cards have been correctly installed and connected to each other by the Programmer card's cable, the main menu is displayed. (Note: the space-bar was pressed to remove the release date and copyright notice from the following display.)

```
ETU: Transfer Configure Blankcheck Quit ?

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
 0UL        4UL
1UL        5UL
 2UL        6UL
 3UL        7UL
```

There are four functions available from the main menu: Transfer, Configure, Blank check, and Quit. Each of these functions will be explained on the following pages.

Your display may differ depending on the select code setting of the Programmer card and the capacity of EPROMs installed in the EPROM card. If more than one Programmer card is installed in the system, all operations will use the "active" Programmer card. If more than one EPROM card is installed in the system, all ETU operations will affect only the EPROM card connected to the (active) Programmer card's cable.

The various functions are activated by typing the first letter of the appropriate operation (for example, C for Configure). Lettercase does not matter. Incorrect letters are ignored except if the program is under stream control. When streaming, incorrect letters will abort the program and the stream file.

All operations can be aborted by typing Shift-Select (SHIFT-EXECUTE) for single character answers or Shift-Select and then Return or Enter (SHIFT-EXECUTE and then ENTER) for multi-character answers.

In stream file operations, answers to optional questions are automatic and are the *first* option given in the prompt. For example:

- For a YES/NO question ending with "(Y/N) ?" – The stream answer is "Y"
- For an ABORT/TRUNCATE question ending with "(A/T) ?" – The stream answer is "A"

Configuration

From the main menu, press C to display the configuration sub-menu. The main menu is replaced with a sub-menu which lets you change the select code, the burn rate, and specify any empty EPROM sockets.

```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ?

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL        4UL
1UL        5UL
2UL        6UL
3UL        7UL
```

The configuration functions are explained next. Pressing Q will return you to the main menu.

Select Code

When only one Programmer card is in the system, it is automatically chosen as the active Programmer card and the select code is properly set.

If you have more than one Programmer card installed in the system and wish to change operations to a different Programmer card, press S for Select code. The following prompt will appear at the bottom of the display:

```
New select code (27) ?
```

The number in parentheses indicates the select code of the currently selected Programmer card. You may either press [Return] or [ENTER] to accept the current select code or type the select code of a different Programmer card. If the select code you type is valid, the display will be updated with the new information. An error message will be displayed if the new select code does not correspond to a Programmer card.

Burn Rate

Pressing [B] will cause the Burn rate to change from SLOW to FAST or from FAST to SLOW (the display is automatically updated).

Note

All EPROMs may be programmed at the slow burn rate. Some EPROMs are not guaranteed to retain the pattern if the faster rate is used. Check the EPROM manufacturer's specifications before using the faster programming rate.

If the FAST burn rate is specified and a location fails to accept the data, the burn rate will automatically be switched to SLOW.

The FAST burn rate programs at 13.1 ms/byte while the SLOW burn rate programs at 52.3 ms/byte. The card circuitry can program both upper (even address) and lower (odd address) bytes in parallel so the effective rate is 13.1 or 52.3 ms/word. Therefore, programming every location in a full set of large capacity EPROMs using the FAST burn rate will take about an hour.

Note that the Burn rate is a global attribute not associated with a particular Programmer or EPROM card.

Empty Sockets

An empty EPROM socket appears to be an erased (blank) EPROM. This condition can not be detected until an attempt is made to program a pattern into such a location.

Pressing [E] allows you to specify which sockets of an EPROM card do not contain EPROMS; the information is used in the calculation of the capacity of an EPROM card. EPROMs must be used in pairs and up to 8 pairs of EPROMs may be used in one card.

```
CONFIGURE: Selectcode Burnrate Emptysockets Qt ? E

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL         4UL
1UL         5UL
2UL         6UL
3UL         7UL

SOCKET (PAIR) NUMBER ?
```

For example, if you answered "7" to this question, the display would be updated as follows:

```
Socket status (UL means eprom pair present)
0UL         4UL
1UL         5UL
2UL         6UL
3UL         7 empty
```

In this manner, you can specify all of the empty sockets. If you make a mistake, re-execute the command with the same socket number; the program will again mark the socket pair as usable.

An error message is displayed if the socket pair number is out of range.

Quitting the Sub-Menu

Quitting the Configure sub-menu will return you to the main menu. Once you have completed the configuration for the active card-pair, the next step is to check for available space in the EPROMs.

Blank Check

Pressing [B] from the main menu will show the used and unused space (according to how many EPROMs you've told the program are on the EPROM card connected to the active Programmer card). A blank EPROM has all of its bits set to binary 1's. Thus, a blank byte would contain the hexadecimal pattern FF.

Unused space will be shown at the bottom of the display. For example:

```
ETU: Transfer Configure Blankcheck Quit ? B

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL         4UL
1UL         5UL
2UL         6UL
3UL         7UL

BLANK CHECK
            0 - 131071 (131072)
```

The number in parenthesis indicates the size of the unused space (in bytes). The two numbers separated by a hyphen indicate the relative address of the unused space within the active card.

The above display indicates that the entire EPROM card is unused. Bytes 0 through 131 071 are blank and the total number of contiguous blank bytes is 131 072.

If no blank bytes can be found on the current EPROM card, the following error message will be displayed:

```
NO BLANK SPACE FOUND
```

Typically, after an EPROM has been programmed, there will be some bytes containing the hexadecimal pattern: FF. These bytes will appear to the program as "blank" and the Blank Check option will list them as follows:

```
address - address (size in bytes)
address - address (size in bytes)
```

The above lines are repeated as many times as required, in groups of 6, with a prompt to press the spacebar to continue between each group. The addresses given are relative to the base address of the EPROM card. The size is likely to be only a few bytes for addresses that actually contain data. (A hexadecimal FF programmed into EPROM looks like a "blank" location.) The last entry is likely to indicate any truly "blank" space. The sockets you've specified as empty are not counted.

Now that the available space has been determined, you are ready to transfer a file or a volume to EPROMs.

Transfer

Pressing [T] from the main menu will prompt you for information about the transfer operation. ETU makes some assumptions to try to help you.

The display will show the following:

```
ETU: Transfer Configure Blankcheck Quit ? T

Programmer card(s) at  27
Active programmer card at select code 27
Burn rate   SLOW
Eprom at address 3932160 for 131072 bytes
Eprom type XX64
Socket status (UL means eprom pair present)
0UL        4UL
1UL        5UL
2UL        6UL
3UL        7UL

TRANSFER OPERATION
Source (ESYS:) ?
```

The ETU program assumes that a volume will be transferred to EPROMs. The volume name in parenthesis is the current prefixed volume. You may accept the volume name by pressing [Return] or you may type another volume name. If you specify both a volume name and a file name then ETU assumes that a single file is to be transferred to EPROMs. If you do specify a different volume or a file, the display will be updated accordingly.

When transferring a volume to an EPROM card, if no "blank" block is found on the EPROM card the following message is given:

```
*** NO BLANK BLOCK ON THIS EPROM CARD ***
```

The program will then display:

```
Start at eprom block offset (0) ?
```

The value in parenthesis indicates the lowest numbered "blank" block. (If every block has been programmed, a zero is displayed.)

Or if a file was specified:

```
Start at eprom byte offset (0) ?
```

For a file, the value in parenthesis is always zero.

If the default value in parenthesis is acceptable, press (Return) to begin the transfer operation. Optionally, you may specify a different block offset or byte offset. See the previous sections on Transferring Volumes and Files for the details about offsets.

If there is insufficient space on the EPROM card for the transfer, the ETU program will prompt:

```
DATA EXCEEDS EPROM SPACE BY xxxx BYTES
Abort transfer or Truncate file (A/T) ?
```

Where xxxx represents the number of excess bytes.

A reply of (A) or (Shift)-(Select) ((SHIFT)-(EXECUTE)) will cancel the operation. A reply of (T) will cause the transfer of only as much data as will fit on the EPROM card. If this happens during the execution of a stream file, the transfer operation will abort and the stream file will be terminated.

A transfer is a two-pass operation. The first pass checks the data and the EPROMs. The second pass actually programs the data into the EPROMs and verifies that it has been stored correctly.

Unless an error occurs, the transfer is automatic from here on.

Check Failure

Check failure is detected during the first pass. The byte to be programmed is matched against the byte on the EPROM card. If the EPROM can not be made to contain the new pattern then a CHECK FAIL results. (An EPROM's "0" bits can not be changed to "1" bits.)

```
CHECK FAIL AT ABSOLUTE ADDRESS aaa
BYTE POSITION bbb FROM START LOCATION
EPROM SOCKET un BYTE nn
```

Where "aaa" is the absolute machine address of the byte which will not program or did not program. The position "bbb" is the byte index (from 0) of the byte in the file. The position is also identified by EPROM ("un" is socket identifier: for example, U1 or L4) and "nn" is the byte offset (from 0) within the identified EPROM.

Burn Failure

If an EPROM fails to accept a byte of data using the FAST burn rate, the utility automatically switches to the SLOW burn rate, updates the display, and attempts to continue.

If the burn rate is already SLOW when a byte fails to program properly, then a "BURN FAIL" occurs. The utility is aborted and a message is displayed. For example:

```
BURN FAIL AT ABSOLUTE ADDRESS 3997696
BYTE POSITION 65536 FROM START LOCATION
EPROM SOCKET 4U BYTE 0
```

If the programming fails exactly on a socket boundary ("BYTE 0" in the example above) check to see if the socket is empty or if the EPROM is improperly installed (bent pins).

Quitting ETU

Pressing [Q] from the main menu will quit the utility and exit to the Pascal Main Command Level.

This concludes the operations of the ETU program. Once the EPROMs in an EPROM card have been programmed, the Programmer card can be removed from the system. (With the power switched off of course!)

Before the File System can recognize an EDISC, a Transfer Method (TM) module must be loaded into the system and a modified version of the CTABLE program must be compiled and executed. (The ETU program has its own driver module and could locate the EPROM card since it was connected to the Programmer card.)

Loading the EPROMS Module

The EPROMS module is supplied on the Pascal 3.0 CONFIG: disc. As with other driver modules, there are two ways to load the module:

- Execute it (with the eXecute command at the Main Level)
- Add it to INITLIB and re-boot

Executing the module "permanently" loads the module, but must be performed every time the system is booted. Adding the module to INITLIB eliminates having to load the module each time you re-boot the system.

Adding the EPROMS Module to INITLIB

If you have two disc drives, the creation of the new INITLIB is relatively simple. If you only have one disc drive, you will need to create a memory volume large enough to hold the new library (about 200 Kbytes).

To create a memory volume, press [M] from the Main Command Level. You will be prompted for the number of 512 byte blocks (answer 400) and the number of directory entries (answer 8). If you are not familiar with memory volumes, see the Memory volume command in the "Main Command Level" chapter of this manual.

1. Initialize a disc, and then use the Filer's Filecopy command to copy the BOOT: disc onto this disc. Since you will be storing the new INITLIB on this new BOOT: disc, you can Remove the existing INITLIB file from the disc. Since the old INITLIB was probably not the last file on the disc (and the new INITLIB will probably be bigger than the old), you should Krunch the disc.

   (It is a very good practice to create a new BOOT: disc rather than modifying your present BOOT: disc. That way you can always return to where you are, no matter what happens to the new disc.)
2. Invoke the Librarian. This is done by pressing [L] from the Main Command Level. If the Librarian is not on-line, insert the ACCESS: disc and try again. Remove the ACCESS: disc once the Librarian has loaded.
3. Insert the *old* BOOT: disc into Unit #3 and the *new* BOOT: disc into Unit #4. (If you are using a memory volume, the memory volume will be the "blank disc". Use whatever unit number you assigned to the memory volume instead of Unit #4 for the remaining steps.)
4. Now use the Librarian to create the new INITLIB.
   a. Press [I] and type `#3:INITLIB.` and [Return] or [ENTER] to enter the Input file. You must include a trailing period to prevent the Librarian from appending the `.CODE` suffix.

      When the Librarian finds the input file, the display will show the name of the first module in the file. You should see the module named KERNEL. If you have a printer, you can press [F] to list all of the modules in INITLIB.

      The EPROMS module can be inserted anywhere after the IODECLARATIONS module but before the module named LAST (it must also precede module BUBBLE, if that module is present). In this example, the module will be included as the next-to-last module in the new INITLIB.
   b. Press [O] and enter `#4:EPLIB.` as the Output file. Again, a trailing period prevents the `.CODE` suffix from being appended to the file name.

      (This disc must *not* be removed until you have finished creating the new EPLIB file.) If you are using a memory volume, use the unit number of the memory volume.

c. Press [E] to enter the Edit mode. You should now see this prompt (in the middle of the screen):

```
F  First module: KERNEL
U  Until module: (end of file)
```

d. Press [U] enter LAST as the Until module. You can now transfer all modules in the file up to (but not including) module LAST by pressing [C].

e. When the preceding transfer is complete, press [A] to append a module to the EPLIB Output file. The Librarian prompts with `Input file:`. Put the CONFIG: disc, or whichever disc now contains the EPROMS module, in Unit #3 (*not* #4, which must not be removed). Enter `#3:EPROMS,` as the Input file specification.

f. The Librarian now prompts with `Enter list of modules or = for all.` Enter = to specify all modules. After the EPROMS module has been transferred to the EPLIB library, the Librarian prompts with `Append done, <space> to continue.` If you removed the BOOT: disc (or the one that contains the INITLIB Input file) to put in the CONFIG: disc, replace the BOOT: disc now *before* pressing the spacebar to answer the prompt.

   (If you removed the BOOT: disc in #3: and did not replace it before pressing the spacebar, you get the following message: `cannot open '#3:INITLIB', ioresult = 10`. In such case, don't worry. Remove the CONFIG: disc and insert the BOOT: disc, then press [I] and enter `#3:INITLIB,` as the Input file. Press [E] to return to Edit mode, and go back to where you were previously by pressing [F] and entering LAST as the First module. Proceed with step g below.)

g. Press [T] to transfer module LAST to the EPLIB file (if you got the error described in the preceding step, press [C] instead of [T]). Then press [S] to stop editing and [K] to keep the file.

h. You should now verify that the EPROMS module was indeed copied to the Output file. Press [I] and enter `#4:EPLIB,` as the Input file. Press the spacebar repeatedly to scan through the modules in the new library file. If you have a printer, press [F] to get a File Directory listing.

i. If all modules are present, then press [Q] to Quit the Librarian.

5. If you have been using two discs, use the Filer to Change the file named EPLIB (on the new BOOT: disc) to INITLIB. If you used a memory volume, remove the old BOOT: disc from Unit #3 and insert the new BOOT: disc; then use the Filer to Filecopy EPLIB from the memory volume to the new disc, changing the file name to INITLIB in the process.
6. Re-boot the computer, which installs the new INITLIB containing the EPROMS module.

To make the EPROM card(s) available to the File System as mass storage units, the CTABLE program must be modified to reserve an entry in the Unit Table.

CTABLE Modifications

The Pascal 3.0 CTABLE program, supplied on the CONFIG: disc, contains a "template" for EPROM cards. You can either use the Editor's Find command to search through all occurrences of the EPROM token until you find the template, or Jump to the end of the program and scroll up until you see the EPROM template shown below.

```
$if false$ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

To activate the template, change $if false$ to $if true$ as shown in the following example:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
$end$
```

The template assigns the lowest addressed EDISC to Unit 42. It should be noted that this unit number is also the default for Bubble cards and may have to be changed to some other unit number more appropriate to your peripheral configuration.

EDISCs are recognized according to their relative addresses in the ROM address space of the system. The EDISC with the lowest address is assigned sequence number 0, the second lowest is assigned sequence number 1, and so on.

If you have more than one EDISC, your template might appear as follows:

```
$if true $ { EPROM DISC }
  {watch for conflicting uses of unit 42}
  tea_EPROM(42,primary_dam,{ sequence number } 0);
  tea_EPROM(27,primary_dam,{ sequence number } 1);
  tea_EPROM(28,primary_dam,{ sequence number } 2);
  tea_EPROM(31,primary_dam,{ sequence number } 3);
$end$
```

To force recognition of an EDISC (or multiple EDISCs), call the procedure TEA_EPROM with the appropriate unit number, DAM identifier, and sequence number.

The connection between unit number and address is made when a CLEARUNIT call is made to the TM. This implies that if the address switches of the EPROM cards are changed, the cards may be assigned different Unit Table entries.

In the Unit Table, the SC field is -1 and the sequence number is stored in the DV field.

Compiling CTABLE

Once the necessary modifications have been made to the CTABLE program, the program should be compiled. (If you do not know how to compile a program, see the Compiler chapter.) Since CTABLE imports several operating system modules, you will need to make the CONFIG:INTERFACE file accessible to the compiler (this file contains the interface text for the operating system modules). To do so, you can either "un-comment" the following compiler option (near the beginning of CTABLE.TEXT):

```
$Search 'CONFIG:INTERFACE'$
```

or add the `CONFIG:INTERFACE` file to the current System Library file. The linking operation is described next.

Linking CTABLE

Once CTABLE.TEXT has been compiled to CTABLE.CODE, the Librarian can be used to create a linked version of CTABLE that will easily fit on the new BOOT: disc.

The following steps assume the program has been compiled as CTABLE.CODE on unit #3. Since the linked version of CTABLE is usually less than 15K bytes, it will be put on the same disc that contains the CTABLE.CODE file and will later be copied to the new BOOT: disc. If you have two drives, you may wish to put the linked (output) file directly onto the new BOOT: disc.

1. Press [L] to invoke the Librarian. You may have to temporarily swap discs if the Librarian is not on-line.
2. Press [I] and enter `#3:CTABLE` as the Input file. The Librarian will add the `.CODE` suffix.
3. Press [H] to specify a new header size. Enter a size of 18. (Setting the header size is similar to specifying the directory size of a disc).
4. Press [O] and enter `#3:TABLE.` as the Output file. The trailing period will suppress the `.CODE` suffix.
5. Perform the actual linking.Syntax:
   a. [L] – to Link. This will update the display.
   b. [D] – to toggle the define source (export text) to NO.
   c. [A] – to transfer all modules.
   d. [L] – to finish linking.
   e. [K] – to keep the output file.
   f. [Q] – to quit the Librarian.
6. Copy the linked TABLE to the new BOOT: disc created earlier. Also copy SYSTEM_P and STARTUP to the new BOOT: disc. The new INITLIB that you created earlier should already be on the new BOOT: disc.

   If you did *not* include the EPROMS module in the INITLIB, the Pascal file system will not recognize the EPROM card until you install the EPROMS module.
7. Re-boot the system using the new BOOT: disc. The File System will now recognize the EPROM card.

EPROM Cards in the File System

After the necessary modifications have been made, and the system re-booted, you can use the Filer's Volumes command to see an EDISC.

For example:

```
Volumes on line:
  1    CONSOLE:
  2    SYSTERM:
  3 #  ACCESS:
  4 *  SYSVOL:
  6    PRINTER:
 42 #  ESYS:
Prefix is - ACCESS:
```

Use the Filer's List command to see the directory. For example:

```
ESYS:                Directory type= LIF level 1
created  7-Jun-82   9.59.37 block size=256
 Storage order
...file name....      # blks      # bytes   last chng

EDITOR                    228        58368   7-Jun-82
FILER                     224        57344   7-Jun-82
FILES shown=2 allocated=2 unallocated=6
BLOCKS (256 bytes) used=452 unused=1 largest space=1
```

You may now use EPROMs as you would any other write-protected mass storage volume. Remember, an EDISC should not be copied to another mass storage volume.

This concludes the EPROM installation and programming information. The remainder of this chapter covers the support modules for EPROMs.

Using DC600 Tapes

This section describes use of the DC600 Streaming Tape Drives, such as the HP 9144, for mass storage operations. If you have one of the Command Set '80 Series Disc Drives, you may also have a DC600 tape cartridge drive integrated into the machine for backup.

Tape Drives Supported

The currently supported DC600 Tape and CS80 Disc/Tape Drives include the following HP products:

- HP 9144
- HP 7908
- HP 7911
- HP 7912
- HP 7914

Tape Lengths

There are two lengths of DC600 tapes: 150 feet and 600 feet; these tapes have capacities of 17 and 67 Mbytes, respectively. Both tapes can be directly accessed by the Pascal File System.

Tape Access Methods

The Pascal system provides two methods of computer-controlled tape access. The first is a utility program with capabilities similar to the integrated disc/tape product's "switch" backup. The Operating and Installation Manual that came with the product describes a method of off-line "switch" backup, involving the use of *save* and *restore* switches located on the tape drive itself. While these switches do provide full-volume image backup capability, they are intended for service-personnel usage only.

The second method is "direct" access to the tape with the Pascal File System, a method which can be used for selective backup of files and logical volumes, even those not on a CS80 disc.

CAUTION

THE DC600 TAPE DRIVES ARE INTENDED FOR USE AS STREAMING DEVICES. THUS, USING THESE TAPES FOR DIRECT ACCESS AND SELECTIVE BACK-UP, ALTHOUGH SUPPORTED, MAY CAUSE ACCELERATED WEAR OR DAMAGE TO THE TAPE DRIVE AND TAPE. IN OTHER WORDS, USE THESE TAPES ONLY FOR LIMITED BACK-UP AND EMERGENCY PURPOSES, NOT FOR NORMAL FILE SYSTEM CALLS IN USER PROGRAMS OR AS PART OF A BOOT SEQUENCE.

Using the Tape Backup Utility

The Tape Backup Utility (TAPEBKUP.CODE) is a program that enables you to copy the complete image of a disc onto a tape, or vice-versa. The utility also provides operations for certifying tapes and verifying the readability of either discs or tapes.

It is important to note that the utility only provides for complete image backup; it does not provide for selective file or volume backup. A limited amount of selective backup is available by using the Pascal file system for "direct" access to the tape.

Concepts and Terminology

Single and Dual Controllers: With the CS80 integrated disc/tape products, the standard option is for the disc and tape drives to share a common controller. The disc is unit 0; the tape is unit 1. One of the features of the shared-controller product is its ability to transfer data directly from unit 0 to unit 1 or vice-versa, without having the data travel through the host computer. This utility was written specifically to support this mode of operation, as it is the most efficient method for complete backup.

There is an option for the integrated disc/tape products where the disc and tape drives each have their own dedicated controller. Each controller has a separate HP-IB port and bus address, and no logical association with the other one. As a result, the "switch" backup capability is not available with this option. Likewise, *this utility does not support the dual controller option.*

Source and Destination Mis-matches: To be consistant with the product's built-in "switch" backup capability, the utility's Medium-copy operation allows all combinations of source and destination sizes, even those which might seem illogical. Thus, if you have more than one of the disc drives in this family, be sure to mark the type of drive which is backed up on each tape. For instance, if you were to restore a tape backup of a 7908 onto a 7911, much of the 7911 would be inaccessible until you re-Zero'ed it appropriately.

Tape Certification: Tape certification is a procedure very similar to hard disc initialization. Even though the tape comes pre-formatted from the manufacturer, it needs thorough testing, with a possible sparing of bad blocks, before it is ready for use. The addresses of spared blocks are entered into a sparing table and those blocks are never used again. While the tape certification process is somewhat lengthy, tapes usually need to be certified only once during their lifetime. Tapes can be purchased that are already certified.

Tape Auto-sparing: Any time problems occur in the reading of a tape block, the tape controller will record this fact on the tape's permanent log, and then automatically spare out the troublesome block during the next write operation to it. This way, the tape actually tends to get better with usage; slightly marginal blocks that may have escaped detection during certification can be spared later. Note, however, that if a tape is re-certified, the previous sparing information is lost, and all defective blocks will have to be re-discovered.

The utility may in certain instances print "Tape certification in progress", and then almost immediately print "Tape certification completed". In this case, the tape was determined to already be certified, so it was *not* re-certified; it merely went through an optimization of its sparing tables.

Tape Unload Sequence: A loaded tape must go through a logical unload sequence before the tape drive will allow you to physically eject it. A tape unload sequence can be *initiated* either by the front panel *UNLOAD* switch or by the utility. Either way, the tape will then go busy for some period of time, to position it for unloading and to update its permanent logs. A minute or two later, you may hear a buzzing noise made by the tape drive heads as the sequence completes and the busy light extinguishes. You may now physically eject the tape.

When the utility prints "Tape unload request completed", it means that the request to the tape drive to *initiate* the unload sequence has completed. You will have to wait for the unload sequence itself to complete before you will be able to eject the tape.

Verification: Verification is a read without the transmission of data back to the host. The device still does its internal data integrity checks, although it usually inhibits the automatic retry mechanism employed by normal reads. In a verify, the data is not actually compared to anything; the device merely verifies that it can read the data correctly.

To Verify or Not Verify a Tape: Explicit verification of a tape takes as much time to do as normal reads or writes. Thus, in deciding whether to verify or not, you must weigh the time it takes to do the verify versus the extra assurance provided by it. With the present series of integrated disc/tape drives (i.e.,7908, 7911, 7912, and 7914), tape verification *is* recommended.

With the stand-alone HP 9144 Tape Drive, however, the drive incorporates a special read-after-write head, which allows verification of the readability of the data *as it is being written.* With these drives, explicit verification is *not* recommended, although it can still be performed.

If a Disc Doesn't Verify: If a disc gives trouble verifying, the recommended procedure is to save its contents to a tape if desired, then re-initialize it using MEDIAINIT.CODE found on the ACCESS: disc. MEDIAINIT will perform a two-pass error rate test on the entire disc, and then intensively test further any blocks with which the disc controller "remembers" having had trouble. All bad blocks will be spared. After MEDIAINIT completes, the saved contents of the disc can be restored from the tape if need be.

In performing a save of the contents of the troublesome disc mentioned above, the utility may report bad blocks on the source, although not necessarily, since a verify inhibits read retries while a copy does not. In such a case, a best guess of the bad blocks' data would be sent to the tape, and the copy operation would complete. The tape would now contain one or more blocks with corrupted data, but it would "verify" correctly, assuming that it and the tape drive were good. Likewise, after restoring the data back to the freshly-initialized disc, the disc would have the tape's identically corrupted data, and it too would now "verify" correctly.

If a Tape Doesn't Verify: If a tape gives trouble verifying, *do not re-certify it*; merely repeat the write operation to the tape again. The utility always uses the tape in auto-sparing mode.

Specifics on 7914 Backup: To be consistant and fully compatible with the 7914's "switch" backup behavior, the utility

- Always requests two tapes
- Doesn't complain if they are not long tapes
- Doesn't complain if the two tapes do not correspond to each other

Even though rigerous checking is not provided, if you exercise moderate caution you shouldn't have any problems.

With a save, the utility always writes the "first half" tape first, followed by the "second half" tape. With a restore, the utility allows you to insert the tapes in either order; an internal "copy start address" field on the tape specifies which area to the disc to restore it to. The utility also prints out the source and destination start addresses for each copy segment, so that you can detect it if you accidentally restore two "first half" tapes or two "second half" tapes.

How to Invoke the Utility: The utility is quite simple to use. Its user interface is similar to the other Pascal subsystems. The `TAPEBKUP.CODE` utility is delivered on the ACCESS: disc. Like any other program, have the code file on-line and use the eXecute command from the Main Command Level to run the utility. When prompted: "Execute what file ?", type:

`ACCESS:TAPEBKUP` (Return) or (ENTER)

The following prompt appears on your CRT.

```
TapebKup: Medium-copy Verify Certify-tape Quit ?
```

Typing the appropriate letter (M), (V), (C), or (Q) selects the corresponding operation.

The Medium-copy Operation

The Medium-copy operation prompts for source and destination media. You specify the source media by entering the volume specification of one of the logical volumes on the media. For instance, `#11:` is often first logical volume on a multi-volume hard disc. After one of the disc volumes has been specified, you are shown a listing of all the other logical volumes that will be affected. The specification for the tape media is typically `#41:`

Medium-copy confirms that you have not specified the same media twice, and that the two associated drives are on a shared controller. If not, it aborts the operation.

Medium-copy also checks the medium sizes and gives one of two informative messages for the situations where the destination is a tape, and the tape is not large enough to hold the entire source image. If the source is a 7914 disc, in which case the only method of complete backup is with *two long* tapes and appropriate swapping, you are reminded of the fact. If the source is not a 7914 disc, in which case a complete backup *cannot* be performed, you are advised of this situation, one which you should normally avoid!

At this point, the utility will ask:

```
Are you SURE you want to proceed? (Y/N)
```

Confirm your selections, and respond with [Y] or [N].

If the destination is a tape, you are given the option to automatically verify it after the copy completes. As usual, respond with [Y] or [N].

If the destination is the tape and it has never been certified, it will now go through that process. Tapes must be certified in order to support auto-sparing. Note that the "switch" save operation does not automatically certify tapes before writing to them.

The copy now takes place under control of the device itself. It proceeds at a rate of about 35 Kbytes per second, or roughly two Megabytes per minute. At this rate, copies with a 7908 take about eight minutes, a 7911: about 14 minutes, a 7912: a little over 30 minutes, and a 7914: also a little over 30 minutes per tape, or about 65 minutes total. All errors are reported to the CRT. If the destination is a tape and it is not completely filled by the copy, an end-of-file mark is appended to the valid data.

If the destination is a tape and you opted for auto-verification, the verify occurs at this point. Only the data actually written to the tape is verified, so that time will not be consumed verifying the entire tape if data was copied to only a fraction of it.

The utility allows you several options if some error occurs in the above certify/copy/verify segment. This is primarily motivated by the 7914's two tape backup sequence, but it is also a nice feature for the single tape sequences. Specifically, if an error does occur, you may elect to:

- Retry the same segment on the same tape
- Manually change tapes and retry the same segment with a different, supposably better tape
- Ignore the error and proceed, usually to the next segment of a 7914 two-tape sequence
- Abort the entire sequence

Once a segment attempt has been completed, either because there were no errors or because you elected to ignore them, the utility automatically *initiates* the tape unload sequence. If you have a 7914, the utility then prompts you to change tapes, and proceeds with the second tape's certify/copy/verify segment.

Finally, if the destination is a disc, an automatic full-volume verification is performed.

The Verify Operation

The Verify operation prompts for a media specification, which may be either tape or disc. Like the Media-copy operation, you specify the media by giving the volume ID for one of the volumes on the media. The utility then prints out all associated volumes, and asks for confirmation to proceed. Type [Y] or [N].

If the device is a tape, you are also given the option for the utility to automatically initiate the tape unload sequence after the verify. Respond with [Y] or [N].

The verify performed here always covers the entire medium, even if the medium is a tape with file marks embedded in it. In contrast, the optional verify of a destination tape during the Medium-copy operation verifies only the data just copied to the tape.

As the verify proceeds, the addresses of all unreadable blocks are printed to the screen. The verify is considered to have failed if any are encountered.

If you requested the auto-unload option for a tape, and the verify fails, the utility will *not* unload the tape, in anticipation that you will want to take further action with the tape.

The Certify-tape Operation
Providing this operation separately may seem unnecessary since the Medium-copy operation automatically certifies uncertified tapes before it writes to them. However, it has been included in the utility in case you want to certify one or more tapes without having to copy a disc image to each one at this time.

Another use of this operation is to force re-certification of previously-certified tapes. You would want to do this only if you suspect that blocks on the tape had somehow been spared when they were really OK. This might have happened on a tape drive with dirty heads.

The Certify operation prompts for media specification, confirmation of your choice, and the tape auto-unload option, in the same manner as with the Verify operation. In addition it asks:

```
Re-certify if already certified? (Y/N)
```

Normally, you will want to type [N], so that that certification will be done only if the tape has never been certified before. However, if you really want to force a re-certification of the tape, with the resultant loss of any previous sparing information, type [Y].

Quitting the Utility
Simply terminates the utility program.

Using the File System for Direct Tape Access

Pascal 3.0 (and later versions) provide you with the capability of directly accessing the tape like you would with any other mass storage device. If one DC600 tape drive is present, it will be assigned as a single LIF volume, unit #41; a second drive will be assigned #42:. The intent of this capability is to allow you to initialize the tape using MEDIAINIT, then using the Filer, transfer files or volume images to it, list its directory, change its volume name, etc.

You can also use the File System to access the *first* volume of a multi-volume disc image that has been backed-up on tape using TAPEBACKUP. You will not be able to access subsequent logical volumes, nor in general access the second tape of a 7914 backup, without first restoring the image to the disc.

CAUTION

THE DC600 TAPE DRIVES ARE INTENDED FOR USE AS STREAMING DEVICES. THUS, USING THESE TAPES FOR DIRECT ACCESS AND SELECTIVE BACK-UP, ALTHOUGH SUPPORTED, MAY CAUSE ACCELERATED WEAR OR DAMAGE TO THE TAPE DRIVE AND TAPE. IN OTHER WORDS, USE THESE TAPES ONLY FOR LIMITED BACK-UP AND EMERGENCY PURPOSES, NOT FOR NORMAL FILE SYSTEM CALLS IN USER PROGRAMS OR AS PART OF A BOOT SEQUENCE.

When you want to use the tape for selective backup/retrieval versus complete backup/retrieval, you have to be careful how you do it, in order to avoid a couple of common pitfalls. These pitfalls are associated with the inherent characteristics of a streaming tape drive, namely its slow seek times and its inability to start and stop rapidly.

For each file written to the tape the following sequence occurs:

1. A seek is performed to the very beginning of the tape to scan the directory
2. The entire directory is scanned, one block at a time
3. A seek is performed somewhere "out in the middle" of the tape to write out the file body
4. A seek is performed back to the beginning of the tape to update the directory

With this information at hand we now discuss two general rules.

Avoid Large Directories on the Tape

Considering that streaming tapes like these can't stop and start between blocks, but actually coast to a stop, back up, and take a running start at the next block, you can see that scanning a large directory one block at a time will be a painfully slow process. In addition, it accelerates wear on both the tape and the tape drive.

What constitutes a "large" directory? You'll ultimately have to decide, but the following data should aid you in making your decision. On a tape with a LIF directory, the first block will contain the LIF volume label and sixteen directory entries. Each block thereafter can contain thirty-two directory entries. Thus, the logical breakpoints in directry sizes are 16, 48, 80, ... 16+32N. MEDIAINIT and the Filer's zero command default to 80 directory entries; it is generally recommended that you not go above this size.

Avoid Transferring Numerous Small Files

Considering that each seek on the tape may take up to tens of seconds, you can see that if you transfer numerous small files, you will probably spend a high percentage of your time seeking back and forth on the tape, and a very small percentage of your time actually transferring data.

Volume Backup

An excellent way to efficiently backup numerous small files is to keep all of them on a single logical volume of the disc, and then backup the entire logical volume in one operation. Two previously seldom-used capabilities of the Filer are volume-to-file and file-to-volume transfers; they provide the key mechanism.

A volume-to-file transfer uses an entire logical volume as the source and saves its complete image as a single file on the destination volume. For example, from the Filer you type [F] to specify a file copy, then type:

```
V11:,#41:VOLBACKUP [Return] or [ENTER]
```

to save the entire image of V11: to a file named VOLBACKUP on volume #41, the tape. While a volume image is in a file, the files within the volume are inaccessible, at least to the average user. To make the files within the volume accessible again, you have to transfer the volume image back to a suitable volume, which is usually the one it originally came from, but need not be as long as it has enough room.

A file-to-volume transfer uses a single file as the source and restores it as a logical volume on the destination volume. For example, from the Filer you type [F] to specify a file copy, then type:

```
#41:VOLBACKUP,#11: [Return] or [ENTER]
```

to restore the file VOLBACKUP, which we assume is a volume image, to its original place. Note thet whatever was on #11 is about to be completely overwritten, so the Filer warns you of this, and asks for your confirmation before proceeding.

Advantages to Selective Backup and Retrieval

Even considering the known pitfalls, selective backup/retrieval to the tape with the Filer is an extremely valuable capability. Here are some advantages:

- You can backup only the files/volumes which changed since the last backup, possibly saving time and the amount of media required for backup.
- You can use the CS80 tape to backup files/volumes from *any* and *all* Pascal-supported mass storage devices, and not just the associated CS80 disc.
- You can interchange data with other HP machines that support LIF on DC600 tapes.
- A single tape can hold may many revisions of the same file/volume, for instance during program development. All revisions of the file/volume must be named uniquely, of course.

Chapter 20
Porting to Series 300

Introduction

This chapter focuses on one objective: making Pascal programs written for Series 200 computers run on Series 300 computers. This process is known as "porting" programs.

Who Needs this Information?

This chapter is directed toward you if you have existing software for Series 200 machines – programs developed by either someone else or yourself. Therefore, it will be of little or no use to you if you are just beginning to develop software for a Series 300 computer.

Methods of Porting

Here are several methods of porting Series 200 software to Series 300 machines:

- Just load it into a Series 300 computer – with no modifications – and run it.
- Write and run a program that properly configures the Series 300 computer for the program.
- Make your Series 300 computer emulate a Series 200 Model 217 computer (by installing a HP 98546A Compatibility Video Card Set), and then run your *unmodified* Series 200 object code on it.
- Modify your Series 200 Pascal source code, re-compile it on the Pascal 3.1 system, and then run it on a Series 300 computer.

Each method has a slightly different set of requirements for its use, as described subsequently.

Chapter Organization

This chapter is organized according to the above strategies. It consists of the following sections:

- Description of enhancements provided by Series 300 computer hardware
- When and how to just load and run the program
- When and how to use a configuration program
- When and how to use the compatibility card set
- When and how to modify the program's source code

Description of Series 300 Enhancements

Acquiring a general understanding of the enhancements to Series 200 computers provided by Series 300 computers will help you to choose a porting method.

Areas of Change

Series 300 computers have enhancements in the following areas:

- Many choices of processor, display, and human interface boards:
  - Five new displays (including a separate, high-speed display controller)
  - Two new processors: MC68010, and MC68020 (with MC68881 math co-processor)
  - Battery-backed, real-time clock
  - RS-232C serial interface (similar to the 98644 serial interface)
  - 46020 HP-HIL keyboard (similar to keyboards used with Models 217 and 237, but different from other Series 200 models)
- No ID PROM (not all Series 200 Models had this feature)

Areas that Did Not Change

It will probably be comforting to know that if a feature is not listed above (and discussed in this chapter), then it is the same for Series 300 computers as for Series 200 computers.

It may also be comforting to note that Series 300 computers can use most of the Series 200 accessories and peripheral devices. See the *HP 9000 Series 300 Configuration Reference Manual* for a complete list.

Displays

Series 300 display technology is the most visible area of change from Series 200 computers.

All Series 300 computers utilize bit-mapped alpha display technology, which *combines* alpha and graphics. (Only the Series 200 Model 237 has a bit-mapped alpha display; all other models have *separate* alpha and graphics planes.)

The main difference between "non-bit-mapped" and "bit-mapped" alpha displays is most easily described in terms of whether the alpha and graphics planes are on independent planes or are on the same plane.

- With "non-bit-mapped" alpha displays, the alpha plane is *separate* from the graphics plane. You can use the (ALPHA) and (GRAPHICS) keys to turn each plane on. When the alpha display is already on, pressing the (ALPHA) key turns off the graphics display. Similarly, pressing the (GRAPHICS) key while the graphics display is on turns off the alpha plane.
- With "bit-mapped" alpha displays, alpha and graphics are displayed on the same plane; there are no separate alpha and graphics planes.

An effect of bit-mapped alpha is that both alpha and graphics are dominant. In other words, displaying a character on the screen overwrites all pixels within the character cell; the previous contents of those pixels are lost (which may have been graphics). Also, any scrolling/clearing of the alpha screen will scroll/clear the graphics information on the screen, since they share the same display plane.

With Series 300 computers, you may choose from one of five displays: both monochrome and color, each available in both medium- and high-resolution versions. Each of these displays requires a different monitor. (Series 200 computers have only one display available for each model.)

- Medium-resolution graphics displays have a default resolution of 512[1] horizontal by 385[2] vertical pixels with DGL (many of the Series 200 graphics displays had 512 × 390-pixel graphics displays).

  Alpha capabilities of these medium-resolution displays are 26 lines by 80 characters (as opposed to the 25 × 80-character alpha displays of many Series 200 computers). The character font for medium-resolution Series 300 displays is a 10 × 10-pixel character in a 12 × 15-pixel cell. These displays have no blinking mode (except for the alpha cursor), and no half-bright mode.

- High-resolution displays have a default resolution of 1 024 horizontal by 752[3] vertical pixels with DGL.

  Alpha capabilities of high-resolution displays are 48 lines of 128 characters, just the same as on the Model 237. The characters are 6 × 10-pixel characters in an 8 × 16-pixel cell. These displays also have no blinking mode (except for the alpha cursor), and no half-bright mode.

Processor Boards

Two processor boards are available with Series 300 computers:

- Medium-performance boards, which feature an MC68010 processor (10 MHz clock rate).
- Higher-performance boards, which feature an MC68020 processor (16 MHz clock rate) and an MC68881 floating-point math co-processor.

(Series 200 computers have an MC68000 processor with an 8 MHz clock, or a 12.5 MHz clock and "HP-UX memory-management hardware" in products with a "U" suffix, such as an "HP 9836U.")

The 68010 is a 16-bit virtual memory microprocessor with a 32-bit internal architecture, and a 16 Mbyte (24-bit) address space. The treatment of virtual memory and the virtual machine of the MC68010 is extended in the MC68020, a 32-bit microprocessor with cache, 32-bit data and address buses, 32-bit data paths, and a 4 Gbyte (32-bit) linear address space. (Note that only 16 Mbytes of address space are available with the Series 200 and 300 systems, because the virtual memory feature is not implemented in the computer's backplane.)

The MC68020 contains an internal 256-byte instruction cache. Each time the microprocessor goes off-chip to fetch opcodes and data, the cache retains the information. Should the need arise to re-execute a recent instruction sequence, the sequence within the cache may still be valid. In this case, the processor reads the instruction information out of the cache without accessing off-chip resources, thus speeding execution. While the MC68020 is executing from the cache, any other bus masters, such as DMA controllers, are free to use the external buses without halting the processor.

1 All Series 300 displays actually have 1 024 horizontal pixels. However, on medium-resolution displays, pairs of contiguous, non-square pixels are treated by the graphics library (DGL) as one unit in order to make square dots on the screen.

2 Medium-resolution Series 300 displays have 400 vertical pixels, of which only 385 are used as a default by DGL. You can also have up to 400 by disabling the on-screen echo of the type-ahead buffer (set bit 8 of the DISPLAY_INIT procedure's "control" parameter).

3 High-resolution Series 300 displays have 1 024 vertical pixels, of which only 752 are used as a default by DGL. You can also have up to 768 by disabling the on-screen echo of the type-ahead buffer (set bit 8 of the DISPLAY_INIT procedure's "control" parameter).

The MC68020 also has a flexible co-processor interface that allows close coupling between the main processor and co-processors such as the MC68881. The MC68881, which provides full IEEE floating-point math support, can execute concurrently with the MC68020 and usually overlaps its processing with the 68020's processing to achieve higher performance. The MC68881 provides increased performance for floating-point operations, in both speed and accuracy, particularly for the evaluation of transcendental functions.

Battery-Backed Real-Time Clock

The Model 310's processor board and the Model 320's Human Interface board have a built-in, battery-backed, real-time clock. This clock, however, has a limited range compared to the Series 200 real-time clock; its range is March 1, 1900 through February 29, 2000. (Only Series 200 Models 226 and 236 could have optionally installed battery-backed, real-time clocks. This hardware was included with the HP 98270 Powerfail Option, whose main purpose was to provide power during brown-out or black-out situations.)

If your program uses the battery-backed, real-time clock, you may need to modify and re-compile the program's source as described in the subsequent "Modifying a Program's Source Code" section.

Built-In Interfaces

All Series 300 computers have a built-in HP-IB interface, which is the same as the built-in HP-IB interface of all Series 200 computers.

Series 300 computers also feature the following built-in interfaces, which differ slightly from some of their Series 200 counterparts:

- RS-232C serial interface (like the HP 98644 low-cost serial interface).
- HP-HIL keyboard interface (like the one in Models 217 and 237)

Serial Interface

All Series 300 computers have a built-in serial interface. As with Series 200 Models 216 and 217 built-in serial interfaces, this interface is permanently set to select code 9. However, this interface differs slightly from versions of the Series 200 built-in serial interface (which are like the optional HP 98626 serial interface).

Since the goal of the built-in 98644 is to provide a low-cost serial interface, there are no hardware switches that allow you to specify default values for the following parameters:

- Select code (hard-wired to 9)
- Interrupt level (hard-wired to 5)
- Default baud rate (Pascal system sets default to 2400 baud)
- Default line control parameters (Pascal system sets defaults to 8 bits/character, 1 stop bit, parity disabled).

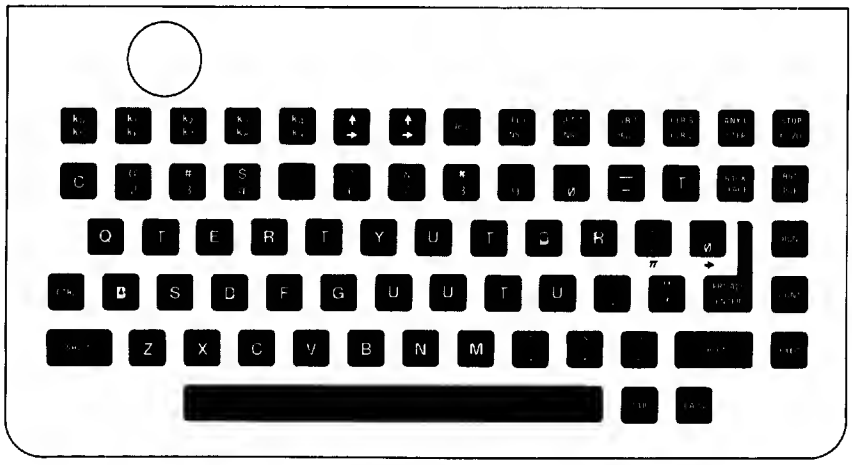
If your program expects any other values for the baud rate and line control parameters, you will have to change them programatically (select code and interrupt level cannot be modified programmatically). See the subsequent "Using a Configuration Program" section of this chapter for further information.

HP-HIL Keyboard Interface

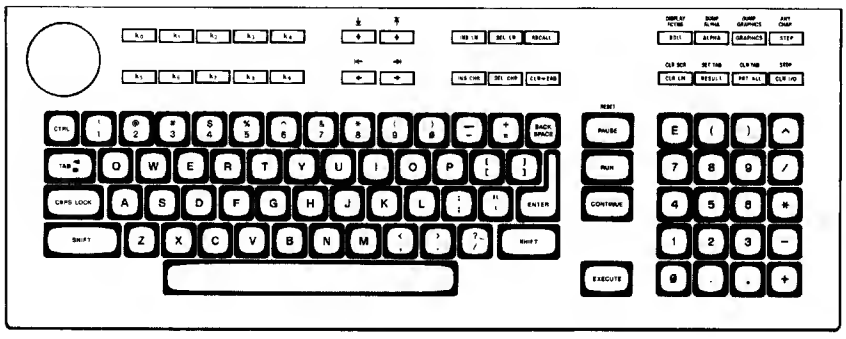
Like the Series 200 Models 217 and 237 computers, Series 300 computers use the HP 46020A HP-HIL (Hewlett-Packard Human Interface Link) keyboard.

If you are porting existing Series 200 software to Series 300 and have already modified it to run on a Model 217 or 237 computer, then you have already made the adjustments necessary for this keyboard. If not, then continue reading this section.

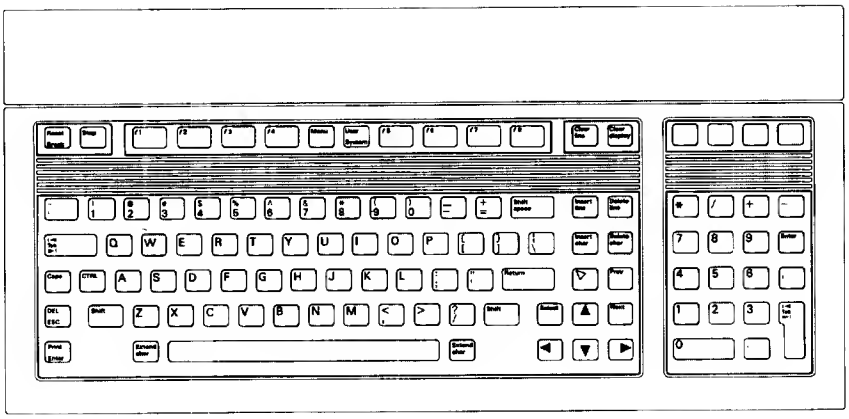
The major human-interface differences between the 98203B keyboard (Models 216, 220, 226, and 236) and the HP-HIL keyboard are in the number and layout of function and system keys.



HP 98203A Keyboard



HP 98203B Keyboard



HP 46020A Keyboard

Note that the Series 300 (46020) keyboard has only eight "function keys," and lacks some of the system keys on the 98203 keyboard. However, the 46020 has *all* of the functionality of the 98203 function and system keys by providing (System) definitions for keys (f1) through (f8). (Press (System) and then (Menu) on the 46020 keyboard to display the system-key labels on the bottom of the monitor screen; default user-key labels are not provided.)

The key mapping is as follows:

- 98203 function keys (k1) through (k8) map into 46020 (User) function keys (f1) through (f8). (The shifted function keys map similarly.)
- 98203 function keys (k0) and (k9) map into 46020 (System) function keys (f1) and (f8).
- Other 98203 system keys map into 46020 (System) keys (see the key labels by pressing (System) and then (Menu) or (Shift) (Menu)).

Also note that the 98203 keyboards can produce some keycodes that cannot be produced with the 46020 keyboard. These key codes are produced by pressing the (EDIT) and (RUN) keys. Thus if the Series 200 program depends upon these keys, the source code must be modified and re-compiled. The topic of trapping key codes with a program is described in the "System Devices" chapter of the *Pascal Procedure Library* manual.

ID PROM

Note that there is no ID PROM available with Series 300 computers, as was the case with many models of Series 200 computers.

Just Loading and Running Programs

This is the most desirable method, since it requires the least amount of work – just load the program into the Series 300 computer, and run it. This section describes when and how to use this method or porting programs.

You can probably port *most* of your 3.0 or 3.01 programs this way.

There are three different actions you can take, depending on who developed your program:

- If HP developed the program, look in the "Operating Systems and Applications" section of the *HP 9000 Series 300 Configuration Reference Manual.* The manual shows which 3.0 or 3.01 applications will run on a Series 300 computer using the 3.1 system.
- If another software vendor developed the program, then that vendor should be able to tell you whether or not it will run on a Series 300 computer. (You can also take one of the two actions listed below.)
- If you developed the program, you can do one of two things:
- Read through the following sections to see whether it requires another porting method.
- Try running it.

Should Problems Arise

If your program will not run on your Series 300 system, then you may want to consider the following:

- Does it meet *all* of the criteria listed in the subsequent sections?
- Is there sufficient memory in the computer?
- Are all the necessary devices and corresponding device drivers installed?
- Have you fulfilled *all* other requirements listed by the software developer?

If the program still doesn't run, then you may want to call the organization responsible for supporting the program (the programmer, the software vendor, or HP).

Using a Configuration Program

This method involves writing a program that configures the system for your program. This section describes when and how to use this method of porting.

Example of Serial Interface Configuration

Here is an example situation for which you could use this method. Suppose your program depends on reading the following parameters from the configuration switches on the 98626-like, built-in serial interface in a Model 217:

- 4800 baud
- 7 bits per character (with 1 stop bit) and odd parity.

However, there are no such switches on the built-in 98644-like interface in Series 300 computers; instead, the Pascal System gives them the following default values:

- 2400 baud
- 8 bits/character (with 1 stop bit), and parity disabled

One solution is to write and run a short program that selects the desired "non-default" baud rate (4800) and line-control parameters (7 bits, odd parity), and then run the program before running the your application program.

This example program changes the "default" parameters by writing to IOCONTROL registers 21 (baud rate) and 22 (line control).

```
program Serial(input,output);
import general_0,general_1;
begin
  ioinitialize;
    iocontrol(9,21,4800);                { Baud rate. }
    iocontrol(9,22,binary('11001010')); { No handshake (bits 7,6)
                                          Odd parity   (bits 5-3)
                                          1 stop bit   (bit  2)
                                          7 bits/char  (bits 1,0)
                                        }
  iouninitialize;
end.
```

You could compile and run this program on the 3.1 system (making sure that the 3.1 IO library is accessible during both compilation and loading[1]). If the RS232 module is not installed, you should install it (with the eXecute command). Then Run the application program, and the serial card will be properly configured.

Another solution is to modify the source program to select these parameters. In such case, you could change the "current" parameters by writing to IOCONTROL registers 3 (baud rate) and 4 (line control). However, if the program later resets the interface with any of the following operations:

- IOINITIALIZE
- IOUNINITIALIZE
- IORESET
- IOCONTROL of registers 1 or 14

or if you press the Stop key (or CLR I/O on the 98203 keyboards), then the values in these registers will be restored to the "default" values currently in registers 21 and 22. See the *Pascal Procedure Library* manual for details on the serial interface registers.

1 The easiest way to ensure this accessibility is to put the LIBRARY: disc on-line and then use the Main Level "What" command to specify the `LIBRARY:IO` file as the System Library (with double-sided media, this is the SYSVOL:IO file).

Using Compatibility Hardware

This method involves installing an HP 98546 Compatibility Video Card Set, which essentially contains the *separate* graphics and alpha planes of the Series 200 Model 217 computer. You can then direct the system to use the compatibility display, enabling you to run some existing Series 200 programs on your Series 300 computer. This section describes when and how to use this method of porting.

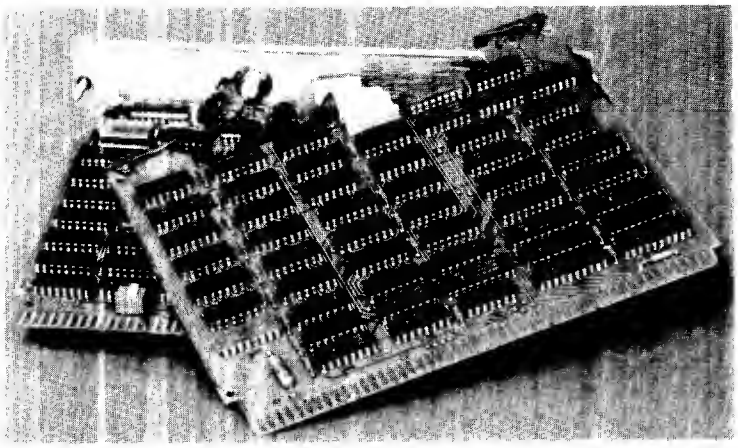
This card set remedies the following situations.

- The program depends on having separate alpha and graphics memory planes.
- The program directly accesses alpha or graphics hardware of a Model 217 or 236A computer (by writing directly into the screen's memory addresses, rather than through a higher-level Pascal or DGL procedure or function).
- The program depends on blinking or half-bright alpha display highlights (characters with codes 130, 131, and 134 through 143).
- The program depends on the Model 217's specific graphics resolution (512 × 390 pixels), alpha display size (80 × 25 characters), or on the registration of alpha and graphics pixels.

This method is required if any of the above statements is true **and** you cannot modify a program's source code (or don't want to). If you have the program's source code, then you may want to instead make the necessary changes in it.

Hardware Description

The card set consists of these two hardware pieces:

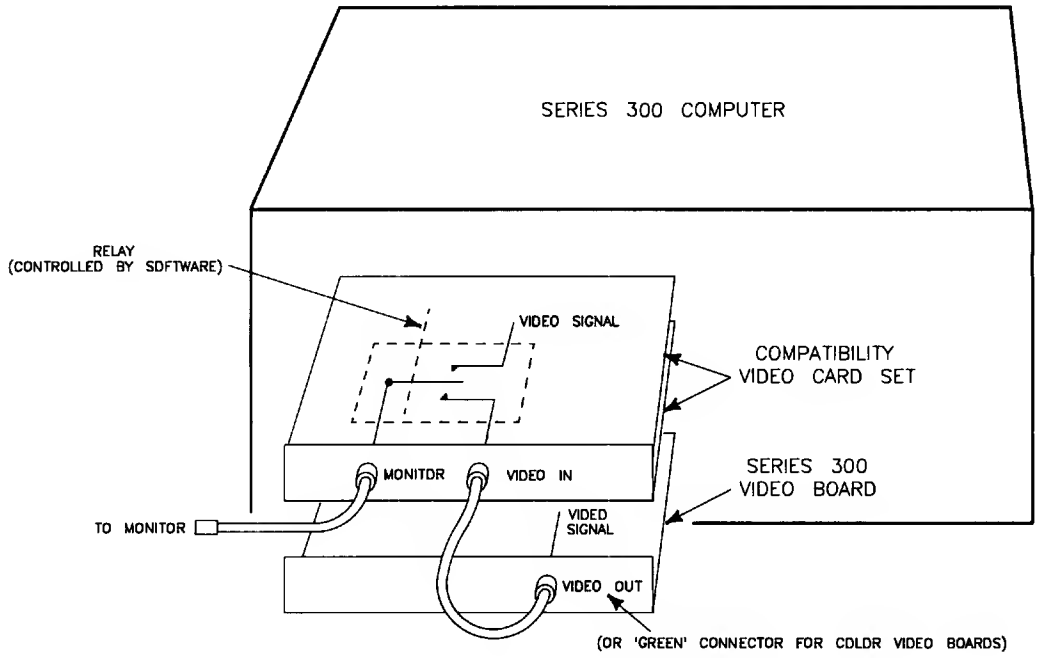


The Compatibility Video Card Set

- The alpha display card is like the existing 98204B display controller card, except for a relay and an additional BNC video connector on the rear panel.
- The graphics card which is identical to the Model 217's graphics card.

The Relay and BNC Video Connectors

The relay on the alpha card is used to switch between using the Series 300's display signal and using the compatibility display's signal.



A Relay Governs Which Display Signal Is Used

Compatibility Video Card Set Capabilities

Capabilities of this card are identical to those of the Model 217. The alpha display is an 80 × 25-character screen with half-bright, blinking, underline, and inverse-video display enhancements. The graphics display is 512 × 390 monochrome pixels.

Configurations Possible

Here are the video-interface/monitor configurations possible:

- **Shared monitor:** The Compatibility Video Card Set and the Series 300 Video Board can share a medium-resolution monitor (monochrome or color).
- **Separate monitors:** The Compatibility Video Card Set can use a medium-resolution monitor, and the Series 300 High-Resolution Video Board can a separate high-resolution monitor (monochrome or color).
- **Single monitor:** The Compatibility Video Card Set can use a medium-resolution monitor (*with* no Series 300 video board or monitor).

Steps in Using this Card Set

Here are the steps you will take with this method:

1. Turn off the computer.
2. Configure and install the Compatibility Video Card Set according to the instructions in its *Installation Note.* Also connect the monitor(s) as described in that note.
3. Boot the system with the disc that uses the desired display hardware.
   a. If you want to use the Compatibility Video Card Set's display hardware, boot the Pascal system using the BOOT: disc. (It is similar to the BOOT: disc supplied with Pascal 3.0 and 3.01, as it contains the same driver modules as in the INITLIB file.)
   b. If you want to use the Series 300's display, boot the Pascal System using the BOOT2: disc. (The INITLIB file on BOOT2: contains the modules CRTC and CRTD, which are the alpha driver for the Series 300 displays and the 98700 Display Controller, respectively. It does not contain the CRT, CRTB, CHOOK, or BAT modules required for Series 200 display hardware.)

Note

Since you are using one monitor for two different displays, a small amount of time is required for the monitor to synchronize with the new display whenever you switch from one display to the other. This will sometimes cause the screen to flicker at power-up or after a soft re-boot. This normally occurs after the `Loading 'INITLIB'` message but before the `Loading 'STARTUP'` message appears on the screen.

Modifying the Source Program

This method involves modifying the program's source code and re-compiling it using the 3.1 system. This section describes when and how to use this method of porting programs.

This method is required for the following situations:

- Programs compiled on the 2.1 or earlier versions of the system.
- The program's object[1] file contains a *linked-in* 3.0 or 3.01 module that is *incompatible* with either the 3.1 system or Series 300 hardware (such as Device-independent Graphics, DGL, modules).
- The program uses any procedures below the level of Workstation Pascal or Procedure Library features (such as the "clock" procedures described in the "System Devices" chapter of the *Pascal Procedure Library* manual).
- The program uses HP 98203 (EDIT) or (RUN) key codes, which cannot be generated by the HP-HIL (HP 46020) keyboard.
- You want to fully utilize Series 300 hardware features which were not present on Series 200 computers (such as use features of the MC68020 processor or MC68881 co-processor).
- The program depends on an ID PROM (this is a memory location that permanently stores the computer's serial number).

If **any** of the above statements is **true**, then you probably need to modify and re-compile the program on the 3.1 system. If you do **not** have access to the source code (or separate object module in the case of the linked modules), then you **cannot** port it – you will have to buy a Series 300 version of the program, if it is available.

Programs Compiled on Pascal 2.1 (or Earlier Versions)

If your program was compiled on the Pascal 2.1 system (or an earlier version), then it will not run on the 3.1 system. You will have to re-compile the source code on the 3.1 system.

If your "pre-3.0" program uses any of the "internal" operating system modules (such as KBD or BAT), you will probably need to re-write the corresponding section of code since these operating system modules were re-designed with the 3.0 system. See the "System Devices" chapter of the *Pascal Procedure Library* manual for details on the new SYSDEVS operating system module.

HP 98203 Specific Key Codes

The 98203 keyboards can generate (EDIT) and (RUN) key codes which cannot be generated by a 46020 keyboard. If your program depends on trapping these key codes, you will need to modify it to use 46020 keys instead. See the "System Devices" chapter of the *Pascal Procedure Library* manual for examples of trapping keystokes with a Pascal program.

1 In this situation, you may not need to modify the source program. You may only need the program's separate *object* file (i.e., the program without the modules linked to it).

Linked-In, Incompatible Modules

An example of this situation is a program that requires 3.0 DGL (Device-independent Graphics) module(s), and the required module(s) are linked to the program (i.e., the modules have been put into the program's object file and linked to it using the Librarian's Link command). Even though you may try to make the program use the 3.1 DGL modules by P-loading them and then running the program, the program will still access the linked-in 3.0 modules. Neither can you remove the linked-in 3.0 modules, since you cannot separate modules in an object file once they have been linked.

To remedy this situation, you will need to have the program's object code and use the Librarian to re-link to it the corresponding 3.1 module(s) that it requires.

Use of Low-Level Procedures

If your program uses any low-level operating system modules, such as SYSDEVS for clock access, then you should probably re-compile it. The reason for this recommendation is that the interface text of these modules may have been modified slightly (and the system does not report any warning message for this type of situation).

Full Utilization of Series 300 Hardware Features

An example of this situation is that programs compiled on a 3.0 or 3.01 system will not make use of MC68881 floating-point math co-processor available on some Series 300 computers.

You can re-compile the program with the COMPILE20 compiler, and the program will make use of this hardware (if installed).

Pascal Compiler Syntax Errors

ANSI/ISO Pascal Errors

1 Erroneous declaration of simple type.
2 Expected an identifier.
4 Expected a right parenthesis ")".
5 Expected a colon ":".
6 Symbol is not valid in this context.
7 Error in parameter list.
8 Expected the keyword OF.
9 Expected a left parenthesis "(".
10 Erroneous type declaration.
11 Expected a left bracket "[".
12 Expected a right bracket "]".
13 Expected the keyword END.
14 Expected a semicolon ";".
15 Expected an integer.
16 Expected an equal sign "=".
17 Expected the keyword BEGIN.
18 Expected a digit following '.'.
19 Error in field list of a record declaration.
20 Expected a comma ",".
21 Expected a period ".".
22 Expected a range specification symbol "..".
23 Expected an end-of-comment delimiter.
24 Expected a dollar sign "$".
50 Error in constant specification.
51 Expected an assignment operator ":=".
52 Expected the keyword THEN.
53 Expected the keyword UNTIL.
54 Expected the keyword DO.
55 Expected the keyword TO or DOWNTO.
56 Variable expected.
58 Erroneous factor in expression.
59 Erroneous symbol following a variable.
98 Illegal character in source text.
99 End of source text reached before end of program.
100 End of program reached before end of source text.
101 Identifier was already declared.
102 Low bound greater than high bound in range of constants.
103 Identifier is not of the appropriate class.
104 Identifier was not declared.
105 Non-numeric expressions cannot be signed.
106 Expected a numeric constant here.
107 Endpoint values of range must be compatible and ordinal.
108 NIL may not be redeclared.
110 Tagfield type in a variant record is not ordinal.
111 Variant case label is not compatible with tagfield.
113 Array dimension type is not ordinal.
115 Set base type is not ordinal.
117 An unsatisfied forward reference remains.
121 Pass by value parameter cannot be type FILE.
123 Type of function result is missing from declaration.
125 Erroneous type of argument for built-in routine.
126 Number of arguments different from number of formal parameters.
127 Argument is not compatible with corresponding parameter.
129 Operands in expression are not compatible.
130 Second operand of IN is not a set.
131 Only equality tests (= and <>) allowed on this type.
132 Tests for strict inclusion (< or >) not allowed on sets.
133 Relational comparison not allowed on this type.
134 Operand(s) are not proper type for this operation.
135 Expression does not evaluate to a boolean result.
136 Set elements are not of ordinal type.
137 Set elements are not compatible with set base type.
138 Variable is not an ARRAY structure.
139 Array index is not compatible with declared subscript.
140 Variable is not a RECORD structure.
141 Variable is not a pointer or FILE structure.
142 Packing allowed only on last dimension of conformant array.
143 FOR loop control variable is not of ordinal type.
144 CASE selecter is not of ordinal type.
145 Limit values not compatible with loop control variable.
147 Case label is not compatible with selector.
149 Array dimension is not bounded.
150 Illegal to assign value to built-in function identifier.
152 No field of that name in the pertinent record.
154 Illegal argument to match pass-by-reference parameter.
156 Case label has already been used.
158 Structure is not a variant record.
160 Previous declaration was not FORWARD.
163 Statement label not in range 0..9999.
164 Target of nonlocal GOTO not in outermost compound statement.
165 Statement label has already been used.
166 Statement label was already declared.
167 Statement label was not declared.
168 Undefined statement label.
169 Set base type is not bounded.
171 Parameter list conflicts with forward declaration.
177 Cannot assign value to function outside its body.
181 Function must contain assignment to function result.
182 Set element is not in range of set base type.
183 File has illegal element type.
184 File parameter must be of type TEXT.
185 Undeclared external file or no file parameter.
190 Attempt to use type identifier in its own declaration.
300 Division by zero.
301 Overflow in constant expression.
302 Index expression out of bounds.
303 Value out of range.
304 Element expression out of range.
400 Unable to open list file.
401 File or volume not found.
403–409 Compiler errors.

Compiler Options

600 Directive is not at beginning of the program.
601 Indentation too large for PAGEWIDTH.
602 Directive not valid in executable code.
604 Too many parameters to SEARCH.
605 Conditional compilation directives out of order.
606 Feature not in standard Pascal flagged by ANSI ON.
607 Feature only allowed when UCSD enabled.
608 INCLUDE exceeds maximum allowed depth of files.
609 Cannot access this INCLUDE file.
610 INCLUDE or IMPORT nesting too deep.
611 Error in accessing library file.
612 Language extension not enabled.
613 Imported module does not have interface text.
614 LINENUM must be in the range 0..65535.
620 Only first instance of routine may have ALIAS.
621 ALIAS not in procedure or function header.
646 Directive not allowed in EXPORT section.
647 Illegal file name.
648 Illegal operand in compiler directive.
649 Unrecognized compiler directive.

Implementation Restrictions

651 Reference to a standard routine that is not implemented.
652 Illegal assignment or CALL involving a standard procedure.
653 CONST, TYPE, VAR, or MODULE cannot follow routine.
655 Record or array constructor not allowed in executable statement.
657 Loop control variable must be local variable.
658 Sets are restricted to the ordinal range 0..8175 (default) or 0..261999 (max).
659 Cannot blank pad literal to more than 255 characters.
660 String constant cannot extend past text line.
661 Integer constant exceeds the range implemented.
662 Nesting level of identifier scopes exceeds maximum (20).
663 Nesting level of declared routines exceeds maximum (15).
665 CASE statement must have non-OTHERWISE clause.
667 Routine was already declared FORWARD.
668 FORWARD routine may not be EXTERNAL.
671 Procedure too long.
672 Structure is too large to be allocated.
673 File component size must be in range 1..32766.
674 Field in record contructor improper or missing.
676 Structured constant has been discarded (cf. SAVE_CONST).
677 Constant overflow.
678 Allowable string length is 1..255 characters.
679 Range of case labels too large.
680 Real constant has too many digits.
681 Real number not allowed.
682 Error in structured constant.
683 More than 32767 bytes of data.
684 Expression too complex.
685 Variable in READ or WRITE list exceeds 32767 bytes.
686 Field width parameter must be in range 0..255.
687 Cannot IMPORT module name in its EXPORT section.
688 Structured constant not allowed in FORWARD module.
689 Module name may not exceed 15 characters.
696 Array elements are not packed.
697 Array lower bound is too large.
698 File parameter required.
699 32-bit arithmetic overflow.

Non-ISO Language Features

701 Cannot dereference variable of type ANYPTR.
702 Cannot make an assignment to this type of variable.
704 Illegal use of module name.
705 Too many concrete modules.
706 Concrete or external instance required.
707 Variable is of type not allowed in variant records.
708 Integer following "#" is greater than 255.
709 Illegal character in a "#" string.
710 Illegal item in EXPORT section.
711 Expected the keyword IMPLEMENT.
712 Expected the keyword RECOVER.
714 Expected the keyword EXPORT.
715 Expected the keyword MODULE.
716 Structured constant has erroneous type.
717 Illegal item in IMPORT section.
718 CALL to other than a procedural variable.
719 Module already implemented (duplicate module).
720 Concrete module not allowed here.
730 Structured constant component incompatible with corresponding type.
731 Array constant has incorrect number of elements.
732 Length specification required.
733 Type identifier required.
750 Error in constant expression.
751 Function result type must be assignable.
900 Insufficient space to open code file.
901 Insufficient space to open REF file.
902 Insufficient space to open DEF file.
903 Error in opening code file.
904 Error in opening REF file.
905 Error in opening DEF file.
906 Code file full.
907 REF file full.
908 DEF file full.

Operating System Runtime Error Messages

Errors detected by the operating system during the execution of a program generate one of the following error messages. The numbers correspond to the value of ESCAPECODE.

0	Normal termination
-1	Abnormal termination
-2	Not enough memory
-3	reference to NIL pointer
-4	Integer overflow
-5	Divide by zero
-6	Real math overflow. The number was too large.
-7	Real math underflow. The number was too small.
-8	Value range error.
-9	Case value range error.
-10	Non-zero IORESULT. (See "I/O System Errors".)
-11	CPU word access to odd address.
-12	CPU bus error.
-13	Illegal CPU instruction.
-14	CPU privilege violation.
-15	Bad argument—SIN/COS.
-16	Bad argument—LN (natural log).
-17	Bad argument—SQRT (square root)
-18	Bad argument—real/BCD conversion.
-19	Bad argument—BCD/real conversion.
-20	Stopped by user.
-21	Unassigned CPU trap.
-22	Reserved.
-23	Reserved.
-24	Macro parameter not 0..9 or a..z.
-25	Undefined macro parameter.
-26	Non-zero IOE_RESULT. (See "I/O Library Errors".)
-27	Non-zero GRAPHICSERROR. (See "Graphics System Errors".)
-28	Parity error in memory
-29	Miscellaneous hardware floating-point error
-30	Bad argument—arcsine/arccosine. Argument>1.
-31	Illegal real number.

I/O System Errors

These are the values found in the system variable IORESULT and the corresponding error message which the system prints out automatically for you.

0	No I/O error reported.
1	Parity (CRC) wrong. I/O driver will do several retries
2	Illegal unit number—valid range is 1..50.
3	Illegal I/O request (e.g., read from printer).
4	Device timeout.
5	Volume went off-line.
6	File lost in directory.
7	Bad file name.
8	No room on volume
9	Volume not found.
10	File not found.
11	Duplicate directory entry.
12	File already open.
13	File not open.
14	Bad input format.
15	Disc block out of range.
16	Device absent or inaccessible
17	Media initialization failed
18	Media is write-protected.
19	Unexpected interrupt
20	Hardware/media failure.
21	Unrecognized error state
22	DMA absent or unavailable.
23	File size not compatable with type.
24	File not opened for reading.
25	File not opened for writing.
26	File not opened for direct access.
27	No room in directory
28	String subscript out of range.
29	Bad string parameter on close of file
30	Attempt to read past end-of-file mark.
31	Media not initialized.
32	Block not found.
33	Device not ready or media absent.
34	Media absent.
35	No directory on volume.
36	File type illegal or does not match request.
37	Parameter illegal or out of range.
38	File cannot be extended.
39	Undefined operation for file.
40	File not lockable.
41	File already locked.
42	File not locked.
43	Directory not empty.
44	Too many files open on device
45	Access to file not allowed
46	Invalid password.
47	File is not a directory
48	Operation not allowed on a directory.
49	Cannot create /WORKSTATIONS/TEMP_FILES.
50	Unrecognized SRM error
51	Medium may have been changed
52	IORESULT was 52.

I/O Library Errors

These are the values and corresponding error messages that may develop when using the I/O library. A call to IOERROR_MESSAGE will generate the appropriate message.

0	No error.
1	No card at select code.
2	Interface should be HP-IB.
3	Not active controller/commands not supported
4	Should be device address, not select code.
5	No space left in buffer.
6	No data left in buffer.
7	Improper transfer attempted.
8	The select code is busy.
9	The buffer is busy
10	Improper transfer count.
11	Bad timeout value/timeout not supported.
12	No driver for this card.
13	No DMA.
14	Word operations not allowed.
15	Not addressed as talker/write not allowed.
16	Not addressed as listener/read not allowed
17	A timeout has occurred/no device
18	Not system controller.
19	Bad status or control.
20	Bad set/clear/test operation.
21	Interface card is dead.
22	End/eod has occurred.
23	Miscellaneous—value of parameter error.
306	Datacomm interface failure.
313	USART receive buffer overflow.
314	Receive buffer overflow.
315	Missing clock.
316	CTS false too long.
317	Lost carrier disconnect
318	No activity disconnect.
319	Connection not established.
325	Bad data bits/parity combination.
326	Bad status/control register
327	Control value out of range

Graphics System Errors

When writing graphics programs, it will be helpful to enclose the main body of the program in a TRY block. In the RECOVER block, test the value of ESCAPECODE. If ESCAPECODE=-27, invoke a graphics function called GRAPHICSERROR. This will return a number which can be cross-referenced with the following list of error messages.

0	No errors since last call to GRAPHICSERROR or INIT_GRAPHICS.
1	Graphics system not initialized.
2	Graphics display is not enabled.
3	Locator device not enabled.
4	ECHO value requires a graphic display to be enabled.
5	Graphics system is already enabled
6	Illegal aspect ratio specified.
7	Illegal parameters specified.
8	Parameters specified are outside physical display limits.
9	Parameters specified are outside limits of window.
10	Logical locator and logical display use same device
11	Parameters specified are outside virtual coordinate system boundary
12	Escape function requested not supported by display device.
13	Parameters specified are outside physical locator limits.

Loader/SEGMENTER Errors

Here is a list of errors that can be generated by the loader or by a program that uses the SEGMENTER module.

100-105	Field overflow trying to link or relocate something.
110	Circular or too deeply nested symbol definitions.
111	Improper link infomation format.
112	Not enough memory.
116	File was not a code file
117	Not enough space in explicit global area
118	Incorrect version number.
119	Unresolved external references.
120	Generated by the dummy procedure returned by FIND_PROC.
121	UNLOAD_SEGMENT called when there are no more segments to unload
122	Not enough space in explicit code area.

Subject Index

a

b

c

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

V

W

Manual Comment Sheet Instruction

If you have any comments or questions regarding this manual, write them on the enclosed comment sheets and place them in the mail. Include page numbers with your comments wherever possible.

If there is a revision number, (found on the Printing History page), include it on the comment sheet. Also include a return address so that we can respond as soon as possible.

The sheets are designed to be folded into thirds along the dotted lines and taped closed. Do not use staples.

Thank you for your time and interest.

MANUAL COMMENT SHEET

Pascal 3.1 Workstation System
for the HP 9000 Series 200/300

98615-90022 May 1985

Update No. _____

(See the Printing History in the front of the manual)

Name: _____

Company: _____

Address: _____

_____

Phone No: _____

fold fold

Programming Experience: _____

_____

System Configuration: _____

_____

_____

Comments: _____

_____

_____

_____

_____

_____

fold fold

_____

_____

_____

_____

_____

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Fort Collins Systems Division
Attn: Customer Documentation
3404 East Harmony Road
Fort Collins, Colorado 80525

Reorder Number
98615-90022
Printed in U.S.A. 5/85

98615-90621
Mfg. No. Only